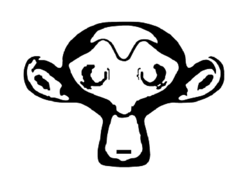
Tecnologías Libres para Síntesis de Imagen Digital Tridimensional

IV Jornadas Blendiberia 2006

Ciudad Real, 7 y 8 Julio



Tecnologías Libres para Síntesis de Imagen Digital Tridimensional

IV Jornadas Blendiberia 2006

Ciudad Real, 7 y 8 Julio

TÍTULO

Tecnologías Libres para Síntesis de Imagen Digital Tridimensional

AUTORES (Por orden alfabético)

Javier Alonso Albusac Jiménez, Fernando Arroba Rubio, Javier Belanche Alonso, Alejandro Conty Estévez, Roberto Domínguez, José Antonio Fernández Sorribes, Luis Fernando Ruiz Gago, Carlos Folch Hidalgo, Miguel García Corchero, Inmaculada Gijón Cardós, Juan David González Cobas, Carlos González Morcillo, Miguel Ángel Guillén, Carlos López Yrigaray, Emilio José Molina Cazorla, Guillermo Vayá.

EDITOR

Carlos González Morcillo

PORTADA Y MAQUETACIÓN

Carlos González Morcillo

IMPRESIÓN

Servicio de publicaciones on-line Lulu.com

ISBN 84-689-9280-1

Nº REGISTRO 06/45890

Este documento fue maquetado exclusivamente con software libre:
GNU/Linux con OpenOffice 2.0, GIMP, Blender y Yafray.



Fotograma de la Película de Animación
"Elephants Dream" realizada con Blender


Don't lose your
monkey head

Tabla de Contenidos

«Los libros que el mundo llama inmorales son libros que muestran al mundo su propia vergüenza»

Oscar Wilde

Fundamentos

Animación

Render

Desarrollo

Prefacio

Este libro que tienes en tus manos es el resultado de varias ediciones de las jornadas Blendiberia, celebradas en Barcelona (2003/2004), Zaragoza (2005) y Ciudad Real (2006). En esta recopilación se han recogido algunas de los documentos realizados para ediciones anteriores de Blendiberia y todas las ponencias preparadas para la última edición, celebrada los días 7 y 8 de Julio en Ciudad Real, coincidiendo espacio-temporalmente con la mayor concentración informática de Castilla-La Mancha llamada Party Quijote 2006.

Para definir qué son las jornadas, citamos algunos párrafos originales de la primera edición, manteniendo los mismos objetivos, concentrando el interés en torno a producciones infográficas y software basado en modelo de código abierto. El estudio técnico de la informática gráfica alcanza su máximo exponente cuando el usuario es capaz de intervenir, modificar y adaptar el software a las necesidades de la sociedad. En cada una de las contribuciones descubrimos la heterogeneidad de dos campos en frenético avance tecnológico: el Software Libre y la Informática Gráfica.

> «Suerte de laboratorio o encuentro eventual de estudiosos a las técnicas de producción y realización de imágenes de síntesis tridimensional y toda periferia creativa que pueda emerger del mismo. El guión lo firma cualquier colaboración ajena, abierta a cualquier persona o colectivo que trate la circulación de la información con libertad de uso, interpretación y distribución.
>
> Aunque no exista un método de trabajo único, ni siquiera su continuidad, el esquema que le da forma es el de una suma de breves piezas que describen un gran puzzle teórico-práctico que va desde el goce abstracto de la programación gráfica a la reivindicación política, aunque ambas cosas se funden en muchos casos.
>
> Lo que sí se puede afirmar es que todas las piezas tienen en común estar hechas desde una libertad absoluta, de la que nos servimos con el objeto y fin de explorar el género de la computación gráfica considerando el usuario como cómplice inteligente, capaz de entender y disfrutar del viaje propuesto. Blendiberia desde su contenido a un grupo de colaboradores que aportan ideas, reflexiones o piezas según su estímulo intelectual. Es obvio remarcar que la mayor parte de las intervenciones giran alrededor de dos herramientas **Blender** y **Yafray**, modelos de software 3D basados en el marco jurídico GNU, pero también se ha querido añadir un acento al desarrollo de proyectos propios que toman como punto de partida o referente la posibilidad del código fuente disponible. »

La maquetación del documento ha intentado conservar la paginación, formato y tamaño de los gráficos originales[1]. Si hubiera algún error en la misma, seguramente el responsible sea el maquetador y no los autores originales del artículo.

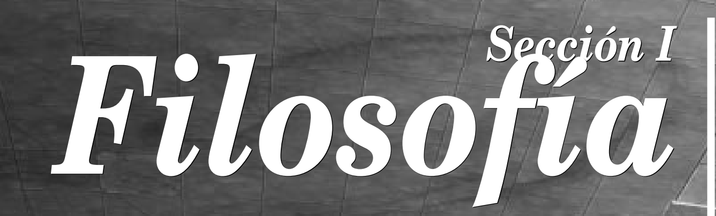
En el diseño de la portada del libro y las *"separatas"* entre secciones se ha utilizado la mascota de blender (Suzanne), una *primitiva* 3D que fue originalmente renderizada por *Javier Belanche*[2].

Esperamos que esta documentación, fruto del esfuerzo de todos los colaboradores de Blendiberia 2006 sea de tu agrado, y si no lo es, siempre podrás enviar tus quejas a `/dev/null`.

Un saludo y *happy blending!*

1 La conversión de algunos artículos ha sido posible gracias a la ayuda de pacientes escribanos (*Luis F. Ruiz*) , auténticos *parsers* de fórmulas LaTeX a formato ODT.

2 Sí, el mono-monocromático es obra suya.



Sección I
Filosofía

Fotograma de la Película de Animación
“Elephants Dream” realizada con Blender


Follow the monkey
MoVeMenT

Creative Commons: entre el copyright y la copia

Javier Belanche Alonso
xbelanch@gmail.com ::



Quería comenzar esta charla sobre CREATIVE COMMONS pidiendo disculpas. Finalmente no voy a hablar demasiado sobre el constructor de licencias que actualmente está teniendo una gran proyección en ámbitos no exclusivos del desarrollo de software.

La última portada de la revista WIRED -responsable en gran medida del nacimiento de CREATIVE COMMONS- regalaba de forma lúdica un CD con 16 canciones con una «nueva y radical» licencia y que sus protagonistas (David Byrne, The Rapture o Beastie Boys, entre otros) forman parte de una iniciativa no carente de dificultades por liberar la producción musical de las actuales restricciones legales y creativas.

Frente a las limitaciones que ha trazado la industria musical, los 13 artistas que suscriben la propuesta han grabado bajo una licencia diferente: la «Sampling Plus License», que nos permite a los usuarios compartir digitalmente las canciones («file-sharing»), pero también nos ofrece la posibilidad de añadir a nuestras producciones comerciales partes o «samples» de estas canciones, siempre y cuando nuestro trabajo sea lo suficiente «diferente» al original para que no resulte una burda copia. En lugar de condenar la habitual práctica de «piratería» que los grandes medios acostumbran a bombardear, se prohibe la falta de creatividad. Está prohibido crear una copia exacta con fines comerciales pero no un «sampleado» cuyos resultados deriven a un trabajo diferente, quizá más excitante o genial. O que comercialmente pueda resultar un éxito inesperado. Y tenemos precedentes. Cuando el grupo «The Orb» consiguió con «Little Fluffy Clouds» un éxito planetario a partir de fragmentos de composiciones del famoso creador minimalista Steve Reich, el mismo artista no supo qué hacer. Finalmente renunció a demandar al grupo «ambient house» considerando la acción misma de «depredar» fragmentos sonoros (no exclusivos de la música, también vale diálogos de series de televisión, ruidos de la calle) y obtener un resultado único, una suerte de collage musical, independiente de las fuentes utilizadas para su creación, aunque el grupo The Orb no las cite.

Es posible la respuesta de Steve Reich se debe al recuerdo de la historia de la música o la historia de las prácticas del surrealismo y dadaísmo, donde el uso y recurso indiscriminado de imágenes tomadas de obras de terceros o incluso adoptando el estilo de creaciones de artistas marginales o tribales era absolutamente aceptado y generalizado. Es obvio que Picasso fundamentó las bases del cubismo en la interpretación personal de las máscaras africanas, de la línea de Ingres y de la investigación pictórica de Cezanne. Incluso André Breton, padre intelectual del movimiento surrealista, invitaba a los creadores plásticos la experiencia de «l'objet retrouvé». Nada de la vanguardia artística del primer cuarto del siglo XX podía ser considerado acto ilegal. En este caso, el artista tomaba sus referencias de obras que, en cierta manera, podemos considerarlas en la actualidad de «dominio público». Y en cierta manera y en otro frente, cuando Walt Disney basó gran parte de su genialidad como animador en la literatura infantil de los hermanos Grimm y otra parte de sus logros gráficos a la atmósfera permisiva de aquella época en la que los artistas disfrutaban de gran libertad para crear obras «derivadas» de otras. Paradójicamente, en la actualidad nadie se atrevería a tomar los personajes clásicos de Walt Disney para una nueva interpretación o, simplemente, para sumar nuevas historias a las ya forman parte de la historia. Sencillamente porque sería acusado de un acto criminal.

¿Por qué en la actualidad se ha llegado a un excepcional y difícil horizonte donde está en juego las libre circulación de las ideas? En unos pocos años, el panorama occidental en cuanto a las reglas de protección y derechos de autor se ha incrementado increíblemente el número de licencias de carácter fuertemente restrictivo. Es más, en ámbitos como la producción y diseño software hemos visto cómo en sus orígenes pertenecía a un campo libre de uso y lectura del código fuente a su privatización en término legales, sin olvidarnos que, en el tramo final, la práctica de patentar código representa la última frontera, el último paso a la exclusión de la cultura a la sociedad. Esta actuación única y focalizada - la exclusión, la privatización fuerte de las ideas -, pero omnidireccional abierta en todos los mercados culturales puede observarse y detectar sus consecuencias en mensajes que manifiestan la inteligencia y la creatividad humana propiedad de un cerebro, de un individuo, conocido popularmente como «artista» o «científico», convirtiéndose descaradamente en marcas registradas de la cultura popular. Así, Bill Gates es a la informática lo que Picasso representa en la evolución de la historia del Arte del siglo XX o, recuperando una historia anterior, lo que Walt Disney representa a la historia de la animación. Este pensamiento no permite la matización. No se permite pensar que lo que generalmente conocemos como «invención individual» es producto de un fenómeno más complejo donde intervienen un número de interferencias o «inteligencias». Ese lema llevado hasta la exasperación conduce claramente a un posicionamiento ultra conservador

que, tal como he comentado antes, defenderá hasta el último aliento una fórmula de estandarización abusiva: un sólo producto, un solo nombre, un monopolio comercial de la cultura.

Frente a esta dictadura de la comercialización y producción exclusiva de la cultura, tenemos otra corriente, un movimiento fractal que, sin entrar en su propia recursividad, lucha abiertamente por la libre circulación de las ideas. Dentro de esta postura se dibujan otras, quizá menos visibles, pero no por eso menos importantes. Vuelvo a incidir en la cuestión de la unicidad creativa. Según estos movimientos de resistencia, la creatividad es propiedad de la humanidad y fomentada en la agrupación de personas con afinidades e intereses comunes y reunidos en torno a una comunidad. El acento de esta revuelta se sitúa en términos de cooperación, de «bazar» en palabras de Eric Raymond. Este movimiento cuestiona abiertamente unas leyes que tuvieron su razón en otro momento histórico, pero que, en la actualidad, han perdido el sentido. La digitalización de la cultura ha trastornado radicalmente el tráfico incesante de la cultura, de las ideas. La digitalización ha transformado completamente el contexto, el marco en el que la tecnología empujó a definir el significado de Copyright como forma legal de proteger las inversiones en la creación de una obra y la creación misma: la copia. En estos momentos, los centros de producción masiva de cultura se han descentralizado. Cualquier persona, desde su casa, puede ser fuente de producción de información; puede copiar, reproducir e incluso compartir sin pérdida en la calidad de la información de los bienes inmateriales con otras personas. Pero, contradictoriamente a lo que parecía una actividad inocua hace unas décadas como el préstamo de un libro a un amigo o grabarle a un vecino una canción, en el día de hoy, estas y otras actividades se han criminalizado con el fin de proteger, según la óptica monopolista del negocio cultural, los derechos del autor. No es extraño que bajo esta atmósfera prohibitiva de la circulación de la cultura hayan cristalizado movimientos como el del software libre, «copyleft» o «all rights reversed». Movimientos que han permitido respirar libertad y que ofrecen de manera legal - y es bastante triste confirmarlo - el poder compartir bienes inmateriales con tu vecino sin riesgo de ser juzgado como un criminal.

Uno de los aspectos más interesantes que contradice los monopolios del comercio cultural y que puede ser motivo de una larga reflexión es el fenómeno japonés conocido por «doujinshi». Todos conocemos la industria del cómic en Japón, «manga». Los japoneses son verdaderos fanáticos de los cómics. Un 40% de las publicaciones en la actualidad son cómics y socialmente cubren todas las edades: desde los más pequeños hasta las personas de edad avanzada. A diferencia de la cultura occidental, el «manga» no forma parte de un uso exclusivo de un grupo o perfil de personas. En España, el cómic siempre ha arrastrado diferentes complejos y prejuicios. Sólo en paises como Francia o Estados Unidos puede decirse

que hay una fuerte tradición y que la sociedad acepta su creación como parte de la cultura social. Sin entrar en más detalles sobre el «manga», lo que realmente me interesa explicar es este fenómeno, «doujinshi», que en términos legales o bajo el paraguas restrictivo del Copyright actual estaría en el terreno de la ilegalidad.

«Doujinshi» son cómics, pero un tipo de cómic especial. Los creadores o dibujantes de «doujinshi» tienen permisos para crear obras derivadas de «mangas» oficiales: no copian, modifican en términos de contribución y desarrollo del original. Un «dounjinshi» puede tomar los personajes de un «manga» y cambiarlos de manera significativa en sus atuendos; o crear nuevas historias para los personajes cambiando su contexto geográfico o histórico; o simplemente continuar historias allá donde el cómic original ha terminado. En Japón, los «dounjinshi» son permitidos en términos legales. No estamos hablando de copias, sino de obras derivadas de otras consideradas «oficiales». En Japón el mercado «dounjinshi» mueve a 33.000 círculos de creadores y dos veces al año se organizan encuentros donde 450.000 japoneses se reúnen para el intercambio o venta de «dounjinshi». Este mercado existe paralelamente al mercado oficial «manga» y el segundo no teme por su florecimiento. En términos occidentales, «doujinshi» equivale a «obra derivada» y, por lo tanto, ilegal.

¿Qué es lo más sorprendente de este movimiento «legal» en Japón? Su existencia. Y al revés de lo que podría pensarse, el mercado «manga» no ha decrecido en absoluto. Al contrario. La existencia de este mercado de aficionados al «manga» favorece el crecimiento de la industria oficial en términos de circulación de información. Esta situación no era muy distinta en los primeros tiempos del cómic americano, donde los dibujantes aprendían los unos de los otros, copiando su estilo, modificándolo o adaptándolo, y del que Walt Disney es un ejemplo. En el día de hoy, la industria del cómic americano, siguiendo la historia, la adaptación o la obra derivada está prohibida. Es más. Personajes como Superman u otros super-héroes tienen bien definidos su rol. Hay cosas que pueden hacer, pero otras que formen parte de la definición de otro super-héroe no. Contrariamente, la derivación en el universo del manga permite una mayor flexibilidad y promiscuidad creativa. Y el mundo de las obras derivadas, tal como ocurre con el «doujinshi», en lugar de restar movimiento al mercado oficial parece ayudar su continuidad.

Retomando la interpretación polarizada de la situación actual, la postura que persigue la mercantilización de la cultura fundamenta su argumento en la protección férrea de los intereses del autor. Recordemos que en la actualidad, cualquier autor -especialmente escritores y músicos- que busque una difusión de su obra en los circuitos comerciales, firmará con un editor o «publisher», casa discográfica o distribuidora un contrato económico en el que se delimita la autoría intelectual de la obra. En 1710,

los derechos de autor en el primer estatuto de Inglaterra se definen como un derecho que equilibra dos partes interesadas: el reconocimiento del autor y, por lo tanto, la protección de la expresión de un contenido, no el contenido del mismo y, en palabras textuales, «animar a los hombres iluminados a componer y a escribir libros útiles». Si bien el autor tiene el monopolio legal de permitir la representación de su obra con su consentimiento, la sociedad ha de poder acceder a la misma con consecuencias creativas. La obra no ha de representar un objeto cerrado a la sociedad y ésta ha de beneficiarse de las creaciones. Para alcanzar este consenso se fijó una caducidad a los derechos de autor que permitiera que la obra formara parte del dominio público después de un número de años. En la ley de la propiedad intelectual española (LPI) y muy similares en otros países, tomaron como referencia su desarrollo las bases redactadas en el Convenio de Berna para la protección de trabajos literarios y artísticos de 1886 donde se diferencian derechos morales y patrimoniales. Los derechos morales protegen indefinidamente la autoría y la integridad de la obra; los derechos patrimoniales hablan del derecho fundamental de la explotación económica de la obra y pueden ser cedidos a un tercero de forma parcial o total de forma exclusiva o no, según el contrato que se fije entre creador y la empresa encargada de la explotación económica de la obra. Los derechos patrimoniales tienen una duración de 70 años después de la muerte del autor, en el caso de la ley española para pasar después a formar parte del dominio público.

La cesión de derechos del autor a un tercero se define mediante un contrato denominado «licencia». En términos amplios, cada licencia define los permisos del uso de la obra, como por ejemplo su explotación o no económica, la creación de obras derivadas, etc; la licencia protege asímismo la atribución o autoría de la obra como el impedimento legal de cualquier modificación de la misma como de la licencia. Por otro lado, la licencia permitirá a un tercero - con exclusividad o no -, tal como hemos comentado anteriormente, a la copia, distribución y representación de la obra.

Dentro de esa definición amplia y general de licencia quiero retomar la «Creative Commons» y así cumplir con el objetivo de esta charla. Fundada en el 2001 y dirigidos por expertos en materia legal de la propiedad intelectual, derecho en la sociedad de la información e informática, la «Creative Commons» viene a ser una suerte de meta-licencia que permita en el panorama digital de la información una ayuda y cobertura legal a los creadores protegiendo su autoría (excepto el caso de dominio público) pero que a un mismo tiempo sea factible la circulación legal de la obra. El lema es evidente: «Creative Commons is a nonprofit that offers a flexible copyright for creative work». Hablar de flexibilidad es sencillamente hablar de ciertas libertades en cuanto a copia, distribución y creación de obras derivadas. El éxito popular que está adquiriendo esta metalicencia es

básicamente por tres motivos: el autor puede definir la licencia que más le interese en apenas tres pasos; una vez definida, el autor tendrá a su disposición tres versiones de la licencia: una versión legible para cualquier persona (Commons Deed), una segunda versión especializada en términos legales (Legal Code) y, finalmente, una tercera que pueda ser interpretada digitalmente por aplicaciones y motores de búsqueda que identifiquen la autoría y los términos de licencia. En segundo lugar comprobamos que, a diferencia de licencias como la GNU, la Creative Commons extiende su cobertura a disciplinas como la música, video, imágenes, texto e incluso educación. Finalmente, Creative Commons ha encontrado traducción legal en un número grande de países, en los que incluimos España y Catalunya, gracias a la colaboración desinteresada de la Universitat de Barcelona.

La creación de una licencia a partir de Creative Commons se fundamenta en la combinación de cuatro condiciones: la atribución, la no-comercialidad, la negación a trabajos derivados y, cuarto, el permiso a trabajos derivados (share-alike), siempre y cuando la obra derivada use y respete la misma licencia que el original. Pero siempre manteniendo dos condiciones previas: afirmar la autoría de la obra y el permiso a terceros a copiar y distribuir la obra con los cuatro condicionantes antes comentados. Permitir, sin lugar a dudas, que la cultura y la circulación de las ideas no esté sujeta a sólo intereses comerciales y revivir el concepto de creatividad como una actuación derivada, no única y genial.

¿Qué pasa en este país?

Alejandro Conty Estévez
aconty@gmail.com ::



Este articulo recoge una serie de impresiones personales sobre el mundo laboral del programador en España. Las directivas que en él se recogen no son evidentemente aplicables como norma. La intención del texto es la de denunciar una serie de aspectos a mi juicio profundamente perjudiciales para el sector.

Como se menciona en "*The mythical man-month*": Recogiendo patatas el individuo más productivo puede tener una relación de 2, como mucho 3 a 1 contra el menos productivo. En el mundo del desarrollo de software, la relación entre el más productivo y el menos productivo puede ser de 10 e incluso 100 a 1. Sin embargo, el más productivo aceptará trabajar sólo por el triple.

Esta ventaja que en principio tienen los empresarios pasa totalmente desapercibida en este país. Hay ya unos cuantos programadores que pueden ser productivos. También hay unos pocos programadores brillantes que podrían, en el entorno adecuado, generar verdaderas fortunas. Pero el programador no es una persona respetada en el mundo laboral. Ni siquiera se le respeta en la calle.

Uno se encuentra programadores cobrando menos que los del servicio de limpieza. No es que éstos deban cobrar menos, es que los programadores deberían cobrar más. Además nos da una idea de como valora la empresa el papel del programador. Al fin y al cabo, en la industria del software son los programadores los que crean riqueza de la nada. Es difícil entender como pueden tener una valoración tan baja. También es sorprendente que los propios programadores lo consientan.

Todos sabemos lo que supone reemplazar a un programador en medio de un proyecto. Puede significar un retraso de 6 meses en el mejor de los casos. El atasco completo del proyecto en el peor de ellos. Pero nadie en la dirección de las empresas se da cuenta. Es más, ni siquiera les importa. En España el 90% de las empresas se dedican a vender humo o a hacer proyectos con la Administración cuyo buen fin preocupa a muy pocos. No es de extrañar que de aquí no salga ninguna empresa espectacular como Skype, Keyhole, o no digamos ya, Google. Debe existir algún motivo.

Trabajando como programador uno se enfrentará a un montón de iluminados visionarios que, sin tener ningún conocimiento, proponen cada día ideas como "*¿Y si hacemos un sistema de telefonía por internet?*". Normalmente estos ataques de creatividad ocurren al día siguiente de que salga por la tele la noticia de que alguien ha hecho lo mismo en alguna parte. A los dos años cuando la idea esté ya implantada podrán decir "*eso se me ocurrió a mí, pero nadie me apoyó*".

Todo este entorno frena a cualquier programador de provecho de hacer nada útil ni revolucionario. A diferencia que en otros paises, las empresas de tecnología no están dirigidas por personal técnico. Están dirigidas por gente de mucho dinero o tipos que ganaron algo de dinero haciendo otra cosa y ahora quieren apuntarse al carro. Nunca saldremos del agujero tecnológico hasta que esto cambie.

En mi corta experiencia como programador he ido desarrollando fuertes alergias a prácticas comunes. Como ahora están de moda los estatutos, he decidido empezar a redactar un Estatuto de los Programadores. Para generar polémica, todo estatuto tiene que tener un preámbulo incendiario que reclame derechos de nación o algo similar. Como no está el horno para bollos, hemos de maquillarlo un poco para que no suene muy radical.

El preámbulo propuesto es el siguiente:

«El mundo de la programación es una realidad prostitucional»

Dicho esto procedemos con el primer y más importante artículo:

Artículo 1: *Toda persona incapaz de realizarlas tendrá, terminantemente prohibido, tener ideas.*

Lleva un poco de reflexión darse cuenta de cuánto beneficio haría una norma como ésta. Primero invito a pensar si podría en algún caso hacer daño a la sociedad. ¿Ha habido alguna vez una buena idea tecnológica o científica que surgiera de mano de alguien incapaz de desarrollarla?

Aunque así fuera (que no es), la idea se hubiera echado a perder por la mala gestión de la misma. Resulta impensable que el jefe de la oficina de patentes donde Einstein trabajaba se presentara un día en el trabajo diciendo:

"*Albert, he pensado que quizás la velocidad de la luz es una constante universal no sujeta a la medición relativa que sufren todas las demás mediciones. Las consecuencias pueden ser muy interesantes, podemos llegar incluso a la energía nuclear. Quiero que dediques todos los días un par de horas a desarrollar esto. Tiene que estar listo en seis meses.*

Los hitos del proyecto son:

1905 - relatividad especial (un primer boceto de la idea)

1915 - relatividad general (desarrollo matemático y generalización)

1950 - centrales nucleares"

Es mucho más probable que lo que ocurriera fuera algo como:

"*Albert, el futuro es desarrollar unas nubes tan densas que la gente pueda volar en ellas y viajar prescindiendo del avión, que es un invento obsoleto. En dos años ya se habrá superado y no podemos quedarnos atrás. Así que deja esos papeles que garabateas todos los días y ponte con esto ya.*"

Las ideas son mucho más que una simple frase. En el mundo de la programación no cuenta sólo la idea, sino la idea de comó llevar a cabo la idea. A los fundadores de Google no se les ocurrió simplemente hacer un buscador. Eso lo hacía todo el mundo por aquel entonces. Se les ocurrieron ideas para hacer uno mucho mejor que el de los demás. Y eso nunca podría habérsele ocurrido a un tipo que reunió unos millones vendiendo camisetas y ahora quiere revolucionar la informática (porque él lo vale).

Un día el jefe llega y dice "*Vamos a hacer un programa de CAD que desbanque al AutoCAD*". Uno se queda esperando por lo que debería venir a continuación. A ver cuáles son esas ideas para un programa de CAD revolucionario. Pero no las hay, eso ya lo hará el programador que contratemos. Al final, si a caso el jefe nos dirá donde colocar los botones de dibujado para que el programa sea bueno de verdad.

Este tipo de casos tendrían que conllevar incluso penas de cárcel. Es por ello que el artículo 1 es fundamental. Las prácticas como ésta llevan a un total desperdicio de tiempo, dinero, paciencia y todo tipo de recursos.

Vamos ahora con el segundo artículo de nuestro estatuto:

Artículo 2: *El uso del PowerPoint y herramientas similares será perseguido y castigado.*

Admito que puede parecer demasiado radical. Cierto es que a la hora de exponer algo a un público, unas transparencias pueden venir bien. Pero en los últimos años, estas herramientas se han convertido en el único objetivo de trabajo de un montón de desorientados. Es uno de los casos donde el medio acaba convirtiéndose en el propio fin. ¿A cuántas presentaciones vacías de contenido hemos de asistir para darnos cuenta?

Una herramienta que permite a cualquiera hacer una presentación consigue implantar la creencia de que todos tienen algo importantísimo que decir.

Ahora mismo la relación señal/ruido en cuanto a presentaciones es bajísima en todas partes. Montones de personas en esta industria llegan cada día al trabajo para abrir inmediatamente el powerpoint y ponerse a hacer la presentación de mañana. Vendemos humo, pero vendemos humo con colorines, animaciones, gráficas dinámicas y sobre todo, mucha negrita e imágenes del catálogo "bussines".

Nadie que quiera decir algo necesita esa parafernalia. Siempre se han hecho "diapositivas", pero se hacían de otra forma, y no se hacía una presentación cada dos días. Lo que estos programas han creado, es una "cultura del powerpoint". El lector puede estar pensando ahora que el hecho de que un cuchillo pueda usarse para matar, no quiere decir que haya que prohibir los cuchillos. Efectivamente, pero si nos fijamos en el artículo 2, nadie habla de prohibir las herremientas. Se trata de perseguir y castigar su uso. ¿Cuántas veces hemos querido decir "Ese gráfico no dice nada, es más, causa confusión, vas a ir a la carcel chaval" ?. Que ocurra alguna vez es aceptable, pero no que ocurra todos los días, miles de veces, en todas las empresas al mismo tiempo.

Y finalemente vamos con el tercer y último artículo por ahora:

Artículo 3: *Se prohíbe el desarrollo de ningún programa que no sea susceptible de ser programado por una sola persona.*

Esto no quiere decir que no puedan trabajar varios en él. Pero un programa que no pueda ser hecho por un sólo programador está, sencillamente, mal pensado. ¿Qué pasa con los grandes sistemas complejos? Antiguamente se tendía al desarrollo de grandes programas monolíticos que tenían un montón de funcionalidades. Enormes sistemas de gestión que controlaban hospitales enteros, sistemas operativos que lo controlaban todo con el mismo código.

Ahora, la tendencia en el mundo ha cambiado. Esos grandes sistemas se desglosan en pequeños programas. Los sistemas de gestión se componen ahora de muchos servicios distintos que se interconectan usando múltiples protocolos. Los programas complejos de por si se reducen al mínimo de funcionalidades y se dejan puertas a la integración con otros programas que hacen otras cosas.

Con esta política de desarrollo, sin duda mejor que la otra, se consiguen programas que hacen unas pocas cosas muy bien. Los programas son más pequeños y por tanto más mantenibles. Incluso los sistemas operativos se dividen ahora en módulos totalmente independientes encargados de tareas específicas. Los clientes prefieren también estas soluciones, porque

cuando quieren añadir algo nuevo al sistema no tienen que reemplazarlo todo. Pueden comprar el sistema de visualización de esta empresa, la gestión de bases de datos de esta otra y el interfaz web de la de más allá.

Pero aquí los empresarios no parecen haberse dado por enterados. Salvo unas pocas muy buenas, la mayoría de pequeñas empresas que quieren crecer se obsesionan con sacar un programa que lo haga todo. Tiene que servir para hablar por teléfono, para grabar cd's, para gestionar los clientes, para mandar un mail, etc ... Sólo las grandes compañías parecen adaptarse a este nuevo modelo de desarrollo. Pero es que en España, éstas llevan tiempo haciendo meras tareas de fontanería informática. En las pequeñas que se dedican a nuevos proyectos se cometen muchos errores. Incluso aunque parezca que los productos que se plantean son muy especializados, al final se le pide al programador que implemente todo tipo de funcionalidades que en ese programa no tienen ningún sentido. El resultado son programas que hacen un montón de cosas, pero ninguna bien.

Hay unos pocos ejemplos de empresas en España que triunfan con productos específicos y de calidad. No sólo por el personal de calidad, además tienen muy claro el objetivo del desarrollo. Por desgracia empresas así no abundan. Pero con pocas que sean, se demuestra que no es imposible.

Aquí acaba este primer boceto del estatuto del programador. Aunque son sólo tres directivas, son cruciales para arreglar el desastre local que sufrimos. Todo el mundo está invitado a intentar completar la lista de artículos en caso de que me haya olvidado de algún aspecto.

Ahora llegamos a la pregunta final. ¿Qué hacer? Lo primero abandonar ese tipo de empresas y convertirnos en su competencia. Muchos programadores deciden muy acertadamente ponerse a trabajar por cuenta propia. No todo el mundo se atreve debido a diversas razones. Entre ellas hay dos muy comunes.

La primera es el miedo a perder la tranquilidad de que vas a cobrar a final de mes. Pero esta tranquilidad no es más que una ilusión. Si en un momento dado la empresa deja de funcionar dejarás de cobrar. Y el problema es que no puedes hacer nada al respecto. Es decir, es posible, muy posible, que un grupo de inconscientes estrelle la empresa contigo dentro. Cuando sólo dependes de ti mismo o de otros como tú al mismo nivel, sí se puede hacer algo. Depende sólo de que hagas las cosas bien.

La segunda es una extraña concepción de la moral. Hay gente que considera incorrecto o desleal irse de una empresa para ir a hacerle la competencia. Hay que deshacerse de esa barrera imaginaria. La empresa no dudaría ni un segundo en prescindir de tus servicios si ya no te necesita. Tampoco verás un céntimo de la riqueza que hayas podido generar más allá de tu mísero sueldo. No sólo se debe hacer por uno mismo, los programas merecen ser bien hechos.

Por desgracia uno no puede decirle a su jefe que va a ser 10 veces más productivo y que quiere cobrar por lo tanto diez veces más. Tomo prestada esta frase de Paul Graham. Entre sus ensayos podemos encontrar un montón de consejos a la hora de lanzar iniciativas de empresa. Al contrario que yo, él es un hombre con experiencia en este campo. Su colección de "bombazos" empieza en el 95 con un sistema de comercio electrónico que vendió a Yahoo y que le hizo inmensamente rico.

Obviamente no estamos hablando del típico que hizo dinero vendiendo ADSL y que ahora quiere hacer programas. Estamos hablando de un gran programador.

La falta de ambición es otro gran problema local. Nadie piensa ni de lejos que puede llegar a hacerse inmensamente rico con un producto. Algunos lo dicen, pero ni ellos ni los que les rodean creen realmente en ello. La gente aspira a ganar un poco de dinero para comprar un BMW y poder presumir en el barrio como "hombre de negocios".

Como advertencia a todo aquél que vaya a salir al mundo laboral de la programación como asalariado, aquí viene lo que encontrará:

- **Salarios bajos.** Dependiendo de la región, el sueldo de principiante va desde los 15000 a los 24000 euros al año. Teniendo ya cierta experiencia se puede llegar con suerte a los 30000. A modo de comparación, en Inglaterra, un sueldo de programador raso es de 50000 libras. Unos 80000 euros. Al igual más o menos que el resto de Europa. No mencionamos ya los sueldos de Estados Unidos por no deprimir al lector. La excusa del precio de la vida ya no es válida. Aunque un DVD cueste más que aquí, en Alemania se puede alquilar un piso por 300 euros.

- **Régimen de desprecio.** Generalmente se apila a los programadores en algún cuarto apartado. Si viene alguien importante, o simplemente, el gerente de otra empresa, en caso de que los llegue a ver será a modo de "Y estos son los programadores". Igual que si se tratara de una jaula de monos. Muchas veces he querido colocar un cartel de "no den de comer a los programadores". Este trato se debe a muchos motivos. Pero de todos ellos, el más sangrante es el miedo. El miedo a que alguien de fuera acabe haciendo una oferta a alguno de los programadores para que se vaya con él.

- **Total anonimato.** No tienes derecho a ningún reconocimiento externo por tu trabajo. El producto es de la empresa. Mejor dicho, del presidente de la empresa. Nadie vendrá a felicitarte por lo que has hecho y mucho menos verás un céntimo de los beneficios. En este país hay empresarios que serían capaces de poner a Miguel Angel a esculpir, hacer millones con sus esculturas y pagarle una miseria de sueldo. Porque al fin y al cabo, el mármol que usa es mío ... ¿Te gustan las esculturas que hace mi empresa?

Si no nos queda otro remedio que trabajar para una de estas empresas porque haya que pagar una hipoteca o cualquier cosa, mi consejo es el siguiente: Jamas regalar ni una sola idea a la empresa. Si se te ocurre cualquier forma de hacer mejor las cosas, lo haces en casa. Lo desarrollas por tu cuenta y ya lo venderás, lo darás como código libre o lo que apetezca. Ya que nos tratan como a ganado, actuemos como ganado. No hacer nada que no se ordene explícita y detalladamente.

No ha sido mi intención mostrar al programador como un individuo extramadamente valioso ni superior al resto de los mortales. El énfasis no se debe a una especial estima por la profesión sino al profundo agujero en el que se haya hundida. Pero existe el riesgo de que como pasó con otros colectivos marginados, se produzca un efecto rebote hacia la sobrevaloración. Convendría no esperar a que la herida fuera demasiado profunda o que todos hayan emigrado hacia otros paises. Este mal no es además exclusivo de la industria del software. Todos conocemos el fenómeno de la fuga de cerebros que sufren todas las disciplinas intelectuales en España. Este caso es sólo uno más de la lista.

Referencias

- Ensayos de Paul Graham:
  `http://www.paulgraham.com/articles.html`
- Sobre PowerPoint:
  - `http://www.norvig.com/Gettysburg/`
  - `http://www.unc.edu/~healdric/Powerpoint.html`

YafRay de la Y a la y

Reflexiones sobre los Proyectos de Software Libre

Luis Fernando Ruiz Gago
luis@yafray.org ::



Cuando uno oye hablar del software libre por primera vez le asaltan un montón de dudas. ¿Quien desarrolla ese software? ¿Por qué se llama *libre* y no *gratis*? ¿¡Gratis!? ¿Por qué se desarrolla y cómo? Si los roedores pudieran roer robles, ¿Cuántos robles roería un roedor? Intentaremos contestar a estas preguntas en el presente documento, siguiendo el ciclo de vida del proyecto en el que me he visto involucrado (en el sentido más amplio de la palabra) en los últimos años: ***YafRay***.

YafRay. Bajo este nombre de lavavajillas se esconde *Yet Another Free Raytracer*, un *raytracer* gratuito y multiplataforma, creado por mi gran amigo **Alejandro Conty Estévez**. Un *raytracer* es un programa que se vale de una técnica llamada *raytracing* para generar imágenes fotorrealistas con un computador.

Su desarrollo comenzó en Julio de 2002. Inicialmente se trataba de un *raytracer* básico para el sistema operativo GNU/Linux. Posteriormente se le fueron añadiendo funcionalidades, una de las más importantes fue el cargador de escenas en un formato propio utilizando XML. De este modo fue posible la exportación de escenas desde programas de modelado para la posterior generación de la imagen utilizando YafRay.

Este hecho produjo un gran interés en comunidades internacionales relacionadas con el mundo de los gráficos tridimensionales (Elysiun -ahora Blender Artists-, Blender.org, Wings3D.org, entre otras), creándose programas que permitían la exportación de escenas desde los modeladores más importantes dentro del software libre como son **Blender** y **Wings3D**.

Como fruto de ese interés se amplió el desarrollo de YafRay a otros sistemas operativos a parte de GNU/Linux. Primero se llevó a cabo la traducción para sistemas operativos Windows. De manera casi simultánea se desarrollaron versiones para Mac OS X.

Actualmente, YafRay se basa en un modelo de *plugins*, en el cual el motor de *render* está contenido en una librería separada del cargador XML. De esta manera, es relativamente sencillo construir cargadores o invocarlo

desde otros programas. A su vez, el motor de render (librería principal) es el encargado de cargar, en tiempo de ejecución, los *plugins* relativos a los métodos de iluminación (*shaders*, luces, etc). Esta modularidad ha facilitado su integración con Blender.

En un principio todo el trabajo de desarrollo recaía sobre Alejandro Conty. A medida que pasaba el tiempo y el interés por YafRay aumentaba, aparecían nuevos desarrolladores y colaboradores. Lo que empezó como un proyecto personal para pasar las horas del verano de 2002, se acabó convirtiendo en un largo proyecto de software libre que involucra a programadores de todo el mundo y con una comunidad de cientos de usuarios.

Antes de entrar de lleno en el proyecto YafRay, me gustaría sentar las bases del modelo de software libre, sin pretender un análisis exhaustivo sobre el ciclo de vida del mismo. No hablaré aquí sobre herramientas de ingeniería o de desarrollo de una manera profunda. El objetivo de este documento, que es el apoyo a la charla del mismo nombre, es introducir al lector en la problemática del software libre, reflexionando sobre las motivaciones de sus desarrolladores, entendiendo el funcionamiento de las comunidades que orbitan alrededor de estos proyectos y sopesando las ventajas e inconvenientes de este modelo.

Existen multitud de estudios a este respecto mucho más precisos que este, en su mayoría elaborados con rigor científico por personas más capacitadas, pero ¡No dejes de leer aún! Aunque no vayas a encontrar aquí una fuerte base teórica de ingeniería del software (libre), puede que te resulte interesante la experiencia del desarrollo del YafRay, cómo se nutre de aportaciones desinterasadas y sobre todo, cómo esta pequeña comunidad de desarrolladores y usuarios se apoya en otras comunidades mayores.

Software Libre

Es de suponer que alguien que lee sobre la evolución de proyectos de software libre sabe a qué se refiere ese tipo de software. Pero como todos hemos sido lectores despistados en alguna ocasión, hay que comentar una vez más que el software libre, según la definición del proyecto GNU, es aquel software que brinda al usuario las siguientes **libertades**:

- **Libertad 0:** La libertad de usar el programa, con cualquier propósito.
- **Libertad 1:** La libertad de estudiar cómo funciona el programa, y adaptarlo a tus necesidades. El acceso al código fuente es una condición previa para esto.

- **Libertad 2:** La libertad de distribuir copias, con lo que puedes ayudar a tu vecino.
- **Libertad 3**: La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. El acceso al código fuente es un requisito previo para esto.

En internet se puede encontrar una inmensa cantidad de literatura, documentación y FAQ's sobre el esta filosofía. El lector que no esté familiarizado con el término debería detenerse en este punto y buscar información al respecto, no sólo para poder seguir con fluidez este documento, sino porque la idea que subyace bajo el concepto es digna de ser meditada. No hay que olvidar que dicha idea se ha convertido para muchos en toda una forma de vida.

Quizás el software libre parezca una idea actual, una vuelta de tuerca a las protestas contra el sistema. Incluso en EEUU han llegado a tachar de actitudes comunistas los principios en los que se basa. Pero realmente, aunque estemos acostumbrados al modelo de software propietario (comercial), en los inicios de la informática era imposible concebir el software como un producto aislado, susceptible de ser vendido. Las licencias por el uso del software y las restricciones para ejecutarlo o copiarlo, sencillamente no existían.

En aquellos tiempos los programas y las máquinas que los ejecutaban estaban íntimamente ligados. No existía el concepto de programa como pieza separada que se tiene hoy. Tampoco había usuarios domésticos, sino que las personas que ejecutaban los programas solían tener muchos conocimientos de programación y por lo general eran científicos e ingenieros.

Entre estos usuarios expertos, lo normal era intercambiar y mejorar los programas, compartiendo sus modificaciones, que a veces recibían el nombre de *hacks* (de ahí la palabra *hacker*, tan mal utilizada en los últimos tiempos).

Uno de aquellos usuarios expertos era **Richard Mathew Stallman** (a veces nombrado por el acrónimo RMS, basado en su nombre de usuario en los computadores del MIT), un personaje a la vez genial y controvertido, imprescindible para comprender el software libre.

Este físico, graduado en 1974 en Harvard, trabajaba en el laboratorio de inteligencia artificial del Instituto de Tecnología de Massachussetts (MIT) desde 1971. En su laboratorio disponían de una impresora que tenía ciertos problemas con la alimentación de papel, de manera que se atascaba habitualmente y no había otra forma de descubrirlo que desplazarse hasta donde estaba.

Richard se puso en contacto con los fabricantes, con la idea de modificar el software que controlaba la impresora y hacer que enviase una señal al atascarse, de forma que no se perdiese tanto tiempo de trabajo.

Sin embargo, éstos se negaron a facilitarle el código fuente, que son como "los planos" de un programa y que hace posible modificar su comportamiento. Este episodio le contrarió mucho e hizo que terminase de consolidarse su idea de que el código fuente de los programas tenía que estar accesible para todo el mundo.

Movido por este deseo, abandonó el MIT en enero de 1984, para iniciar el **proyecto GNU**. GNU es un acrónimo recursivo que significa *GNU's Not Unix*, GNU No Es UNIX, en referencia a que el proyecto busca desarrollar un sistema operativo de tipo UNIX, pero libre.

En sus comienzos, el proyecto GNU se concentró en desarrollar las herramientas necesarias para construir un sistema operativo, como editores y compiladores y en las utilidades básicas para la gestión del sistema. Sobre 1985, Richard Stallman creó la licencia GPL (*General Public License*) como mecanismo para proteger el software libre, sustentado sobre el concepto de *copyleft*. Mediante este concepto, se le da la vuelta a la idea de *copyright*, definiendo las cuatro libertades mencionadas anteriormente.

Uno de los conceptos que se suelen mezclar con el software libre (reconozco que me sucede habitualmente) es el de ***Open Source*** *(código abierto)*. Podemos considerarlo como un error justificable, ya que en la práctica el software *Open Source* y el software libre comparten las mismas licencias. Aunque la FSF (*Free Software Foundation*) opina que el movimiento *Open Source* es filosóficamente diferente del movimiento del software libre.

Apareció en 1998 con un grupo de personas, entre los que cabe destacar a **Eric S. Raymond** (autor de "La Catedral y el Bazar" y de *fetchmail*) y **Bruce Perens** (padre del proyecto *Debian*), que formaron la *Open Source Initiative* (OSI). Buscaban darle mayor relevancia a los beneficios prácticos del compartir el código fuente, e interesar a las principales casas de software y otras empresas de la industria de la alta tecnología en el concepto. Estos defensores ven que el término *Open Source* evita la ambigüedad del termino Inglés *free* en *free software*.

Mucha gente reconoce el beneficio cualitativo del proceso de desarrollo de software cuando cualquiera puede usar, modificar y redistribuir el código fuente de un programa. Esta es la idea subyacente de la obra de Raymond, "La Catedral y el Bazar", que analizaremos más adelante. El movimiento del software libre hace especial énfasis en los aspectos morales o éticos del software, viendo la excelencia técnica como un producto secundario

deseable de su estándar ético. El movimiento *Open Source* ve la excelencia técnica como el objetivo prioritario, siendo la compartición del código fuente un medio para dicho fin. Por dicho motivo, la FSF se distancia tanto del movimiento *Open Source* como del propio término.

Ingeniería del Software

La ingeniería de software es la rama de la ingeniería que crea y mantiene las aplicaciones de software aplicando tecnologías y prácticas de las ciencias computacionales, manejo de proyectos, ingeniería, el ámbito de la aplicación, y otros campos.

Este término es relativamente nuevo, ya que no apareció hasta finales de los 60, cuando ya existían grandes compañías informáticas. En aquella época, una serie de estudios sobre desarrollo de software llegaron a la conclusión de que existía una "crisis del software". Esta crisis tiene los siguientes síntomas:

- El software no es fiable y necesita de un mantenimiento permanente.
- El software se entrega muy a menudo con retrasos y con unos costes superiores a los presupuestados.
- A menudo el software es imposible de mantener, carece de transparencia y no se puede modificar ni mejorar.

Como solución a esta crisis surge la idea de aplicar un proceso de ingeniería para el desarrollo de software. Según la definición del IEEE, la ingeniería del software es "*un enfoque sistemático y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento del software: es decir, la aplicación de la ingeniera del software*".

La ingeniería, no sólo la del software, pretende aplicar con criterio el conocimiento matemático y científico obtenido a través del estudio, la experiencia, y la práctica. En la mayor parte de las ramas de la ingeniería es posible cuantificar con exactitud plazos, recursos humanos, costes y técnicas que lleven a cabo un determinado producto. Pero cuando ese producto se trata de software surgen los problemas, ya que aún no se han encontrado métodos de desarrollo y técnicas que permitan producir software de gran calidad con unos recursos limitados.

A pesar de que la ingeniería del software ha conseguido notables éxitos, no es menos cierto que no ha sido capaz de superar la crisis del software. Hay quien piensa que más que de una crisis, estamos hablando de una

enfermedad crónica. Es una reflexión plausible, más aún viendo como en los últimos años se han retomado viejos caminos bajo nuevas fórmulas de ingeniería. Quien sabe, quizás sean los nuevos modelos de desarrollo los que permitan al software librarse de esta crisis, hasta hoy irresoluble.

Ingeniería del Software tradicional: Software propietario

Dentro del modelo propietario, la propia forma de desarrollar software ha sido quien ha llevado a la ingeniería del software a la crisis. No es una afirmación a la ligera ya que, como afirma Gregorio Robles, "*el formato binario del software, la opacidad en los modelos de negocios, los secretos y barreras comerciales se encuentran entre las principales causas que han imposibilitado estudios cuantitativos y cualitativos a gran escala del software cuyos resultados pudieran ser verificados sistemáticamente por equipos de investigación independientes*".

Es paradójico ver como el propio modelo de desarrollo destinado a evitar que un software robusto y potente pueda ser aprovechado por la competencia, impide la consecución del objetivo primordial: Conseguir un software robusto y potente.

Ingeniería del Software Libre

No hay que entender al software libre como un competidor directo sobre el software propietario. Hay diferencias de base, como las razones motivacionales, casi filosóficas, de los desarrolladores. Tampoco se puede comparar a nivel económico, ya que el software libre sigue sus propias pautas de mercado y desde luego, no suele coincidir con el software propietario en la forma de obtener beneficios. Y lo que es más importante, la forma de producir software es totalmente diferente.

La ingeniería del software no es ajena a todo esto y desde hace unos cinco años viene estudiando este nuevo modelo de desarrollo. En un principio podemos pensar que la propia naturaleza del desarrollo de software libre va a hacer fracasar cualquier intento de aplicación de ingeniería. Por ejemplo, si en el caso propietario la medición de costes es difícilmente cuantificable, en el caso del software libre tendremos que acudir a magos y hechiceros para que nos resuelvan la papeleta. Sin embargo, la transparencia en todos los procesos, la disponibilidad del código y en la mayor parte de los casos de todas las versiones de desarrollo, permiten que podamos analizar profundamente cualquier proyecto.

No debemos caer en la tentación de reducir el análisis a un simple punto de vista estadístico. Hay muchas características dentro de un desarrollo libre que caen dentro del campo de lo sociológico. Analizar la comunidad de usuarios y desarrolladores que orbita alrededor de un proyecto de software libre nos dará una gran información sobre la naturaleza de las decisiones, fallos o aciertos, que se hayan tomado durante el desarrollo. Nuevamente el carácter abierto de estas comunidades nos brinda la posibilidad del análisis, ya que las bases del conocimiento suelen estar contenidas en foros y listas de distribución.

Las comunidades son el auténtico motor del software libre. Habitualmente las forman indistintamente usuarios y desarrolladores. No suele estar muy clara la distinción entre ambos roles ya que, incluso los usuarios que no generan código, se convierten en parte del proceso de desarrollo realizando una de las labores más tediosas en la producción de software: las pruebas. Este es uno de los grandes aportes del software libre, puesto que los usuarios influyen, directa e indirectamente, en el desarrollo. La evolución de la comunidad pues, forma parte de la evolución del propio software.

Veamos el caso de YafRay. En un principio no era más que un proyecto personal de su autor, planteado como programa independiente y con unas pocas funcionalidades. Esto suele ser una constante en el software libre, donde los proyectos surgen por acciones puramente personales. Normalmente el desarrollador decide publicar el código cuando ve limitada su capacidad, no necesariamente en conocimientos, sino que también se suele deber a falta de tiempo para realizar todo el trabajo. En el caso de YafRay el código se liberó desde un principio, ya que no pretendía ser más que un experimento del autor con el fin de aprender y aplicar conceptos sobre *raytracing*.

Los usuarios de la comunidad fueron los que convirtieron el proyecto en algo más serio y de alguna forma, marcaron el camino sobre qué funcionalidades deberían ser implementadas primero. Además crearon la necesidad de integrar este motor de *render* al modelador Blender, ya que en aquellos momentos Blender sufría de muchas carencias en el proceso de renderizado. Esa “presión” de la comunidad llevó a YafRay a lo que es ahora, un *raytracer* tan funcional que puede ser (y de hecho, es) utilizado profesionalmente.

A día de hoy tiene una alta integración con Blender y Wings3D, y el proyecto a crecido tanto que en estos momentos se está llevando a cabo una fase de rediseño, con el fin de solventar ciertas problemas debidos a este nuevo rumbo que la comunidad ha querido que YafRay tomara. Además, el nuevo rediseño se está pensando para que atraiga a más desarrolladores a la comunidad, y los que ya existen se sientan más involucrados tomando decisiones “desde abajo”.

Pero... ¿Quienes son los miembros de las comunidades? Al hablar de comunidades de software libre nos suele venir a la cabeza un grupo de *frikis* discutiendo sobre *Star Wars* y las posibles consecuencias de enfrentar a *Batman* y a *La Masa*. Es una injusticia que a las personas que amamos el software libre se nos encasille de esta manera, así que es hora de desmitificar la leyenda. De todos modos, espero que quede claro que *La Masa* patearía el culo a *Batman*.

Las comunidades de software libre son actualmente el objetivo de estudio de economistas, psicólogos y sociólogos. Se trata de una nueva forma de comunidad virtual, con reglas y jerarquías propias, y en muchos aspectos totalmente diferentes a las que conocemos en la sociedad tradicional. Saber quién forma parte de estas comunidades y cuáles son sus motivaciones es determinante para entender este movimiento. En el caso de los desarrolladores las incógnitas son aún mayores, debido a que en una sociedad de consumo es difícil entender por qué las personas dedican su tiempo libre a una actividad, aparentemente costosa y con beneficios directos prácticamente nulos.

Hay una gran cantidad de recientes estudios científicos que intentan definir el perfil del desarrollador de software libre. En el documento "Introducción al software libre" de la UOC podemos leer:

> *«Los desarrolladores de software libre son generalmente personas jóvenes. La media de edad está situada en torno a los 27 años. La varianza de la edad es muy grande, ya que el grupo predominante se encuentra en una horquilla que va desde los 21 a los 24 años, siendo la mediana - el valor que aparece con mayor frecuencia - los 23 años. Es interesante observar cómo la edad de incorporación al movimiento de software libre tiene sus máximos entre los 18 y 25 años, siendo especialmente pronunciada entre los 21 y 23 años , lo que equivaldría a la edad universitaria. Esta evidencia contrasta con la afirmación de que el software libre es cosa principalmente de adolescentes, aunque su presencia es evidente (alrededor de un 20% de los desarrolladores tiene menos de 20 años). En definitiva, podemos ver como los desarrolladores suelen ser mayoritariamente veinteañeros (un 60%), mientras que los menores de 20 y los mayores de 30 se reparten a partes iguales el 40% restante.»*

En el mismo texto se hace referencia a otros estudios, en los que se llega a la conclusión de que la mayor parte de los desarrolladores, el 60%, tienen pareja e incluso un 16% tiene hijos. Con esto se acaba de desmoronar el mito del desarrollador de software libre como una persona adolescente y aislada, sin más conocimientos ni relación con el mundo exterior que los que llegan a través de su computadora. El hecho de que el perfil del desarrollador coincida con el del universitario medio no debería chocarnos, a fin de cuentas el movimiento del software libre tiene sus

orígenes en ambientes universitarios. Richard Stallman recuerda su trabajo en el *Artificial Intelligence Lab* del MIT en los años setenta, cuando el software compartido se consideraba parte fundamental del proceso:

> *«A nuestro software no lo llamábamos software libre, porque ese término aún no existía, pero es exactamente lo que era. Cuando gente de otra universidad o una empresa querían un programa para hacerlo compatible y utilizarlo, se lo prestábamos con mucho gusto. Si veías a alguien usar un programa desconocido que te interesaba, siempre podías pedir que te dejaran ver el código fuente para así poder leerlo, cambiarlo o desmontarlo para crear un nuevo programa.»*

Sin embargo, otro de los grandes mitos se torna real en las encuestas. El software libre es desarrollado en su mayor parte por varones. La presencia de las mujeres en las comunidades varían entre el 1% y un 3%, compitiendo dúramente por el protagonismo en las gráficas estadísticas con el error de muestreo...

Mientras que es relativamente sencillo saber la distribución de los desarrolladores según su edad, sexo y estatus social, averiguar las motivaciones por las cuales se emplea tiempo y, en ocasiones, dinero, resulta una tarea mucho más complicada. A veces los propios desarrolladores no tienen demasiado claras las motivaciones o, lo que es muy habitual, existe una conjunción de varios factores. Desde luego las encuestas nos sirven para eliminar posibles causas de participación.

La primera causa en ser descartada es la económica porque, aunque es totalmente factible obtener beneficios económicos con el software libre, estos beneficios nunca se podrán obtener de una forma directa por la venta de licencias, como en el caso del software propietario. Aún así, el 50% de los desarrolladores encuestados afirman haber obtenido recompensa económica como causa de su implicación en un proyecto de software libre. Sin embargo, hay muchos que no lo ven tan claro. El propio Richard Stallman, ante la pregunta de qué debe hacer un desarrollador de software libre para **ganar dinero**, suele responder: "*puede trabajar de camarero*". Personalmente opino que quitando la palabra *libre* a la pregunta, la respuesta no varía.

Al ser preguntados por su ocupación profesional, los desarrolladores se definen como ingenieros software (33%), estudiantes (21%), programadores (11%), consultores (10%), profesores de universidad (7%), etc. Es interesante observar como, a pesar de que en el software libre no se aplican técnicas clásicas de ingeniería del software, hay tres veces más de personas que se definen a sí mismos como ingenieros de software antes que programadores.

Una vez descartada la motivación económica se suele recurrir al ego de los desarrolladores. Si el lector estaba pensando en desarrollar software libre buscando la fama, puede ir olvidando la idea. En el documento: "*Free/libre and open source soft-ware: Survey and study - part IV*" de Ghosh, Glatt, Krieger y Robles (2002) se incluyó una pregunta, dirigida a los desarrolladores, para que indicaran a qué colegas de una lista dada conocían, no necesariamente de manera personal. Los resultados fueron determinantes.

La mayor parte de las personas conocían a pesos pesados del desarrollo, personas como Stallman, **Icaza** (*GNOME*) o **Torvalds** (*Linux*), eran conocidas por la práctica totalidad de los encuestados. Ni

que decir tiene que son personas que se dedican a algo más que al desarrollo y que tienen grandes connotaciones filosofico-históricas dentro del mundo del software libre. Lo curioso es que otros grandes desarrolladores que venían en la lista, como **Jörg Schilling** (*cdrecord*), **Marco Pressenti Gritti** (*Galeon*), **Guenther Bartsch** (*xine*) o **Bryan Andrews** (*Apache toolbox*) tenían el mismo grado de popularidad que Martin Hoffstede o Angelo Roulini, personas inexistentes y hábilmente añadidas por los encuestadores para averiguar el margen de error de las encuestas.

Así que, amigo lector, olvide el desarrollo del software libre como vehículo para **obtener fama y fortuna**. A menos, claro está, que sea realmente bueno sirviendo copas y que su aspiración sea obtener la misma fama que las personas que no existen.

Aunque no se han realizado estudios rigurosos, podemos afirmar que el proyecto YafRay encaja en los valores medios comentados anteriormente. Comparte una parte de su comunidad con Blender. Martine Aalbers de la *Amsterdam University* realizó un estudio sociológico entre 746 personas de la comunidad de desarrolladores y usuarios de Blender, entre los que nos encontramos gran parte de los miembros del proyecto YafRay.

Se refrendaron algunos datos anteriormente comentados, como la baja participación femenina. No obstante, en el caso de Blender, la horquilla de edad es mayor (un 85% entre 15 y 35 años), sin duda debido a que existe una gran cantidad de usuarios adolescentes. Sin embargo, sólo un 21% de los encuestados reconocían haber desarrollado parte del software.

Un dato interesante es que el 91% de los encuestados se consideraban usuarios de Blender, lo cual supone que los desarrolladores son, además, usuarios del producto final.

En cuanto a las motivaciones que llevan a las personas a formar parte de la comunidad de Blender, la respuesta obtenida fué la siguiente (ordenada de mayor a menor motivación):

- Entretenimiento. Participar por diversión.
- Altruismo. Querer ayudar a la gente sin obtener nada a cambio.
- Ayuda a la comunidad. Participar porque estas comprometido con la comunidad.
- Reciprocidad. Querer ayudar a la gente y esperar recibir algo a cambio.
- Reputación. Participar para labrarse una reputación.
- Mejoras en el software. Participar por querer o necesitar mejoras en el software.
- Recompensa económica. Participar para ganar dinero.

Como se ve, las motivaciones que se pueden considerar más "egoístas" son las que están menos presentes. De hecho, en el propio estudio se resalta la inesperada posición del altruismo entre las motivaciones.

La Catedral y el Bazar

En 1997 Eric S. Raymond escribió el primer documento que trataba de describir las características de los modelos de desarrollo de software libre, comparándolas con el modelo propietario. El artículo, como hemos mencionado antes, se titula "La catedral y el bazar" y se ha convertido en uno de los más conocidos y criticados del mundo del software libre, hasta el punto que para algunos supone el comienzo de la ingeniería del software libre.

Raymond establece una analogía entre el modo de construir las catedrales medievales y la forma de clásica de producir software. En ambos casos existe una distribución de tareas diferenciada, en la que el diseñador está por encima de todo, controlando el desarrollo de la actividad. También la planificación está totalmente detallada desde el principio, marcando las funciones de cada uno de los participantes y las técnicas que han de utilizar para llevar a cabo su labor.

Dentro de esta idea de catedral no sólo está incluido el desarrollo clásico de software propietario. Raymond encuentra que proyectos de software libre como GNU o NetBSD están fuertemente centralizados, ya que unas pocas personas son las que realizan el diseño e implementación del software. Si una persona quiere entrar a formar parte del equipo de desarrollo, los supervisores del proyecto le asignaran un rol y una tarea. Además, este tipo de proyectos tiene unas entregas del software *releases* espaciadas en el tiempo siguiendo una planificación bastante estricta.

En contraposición al modelo de la catedral está, según Raymond, el modelo del bazar. En un bazar no existe una autoridad que controle todos los procesos que se están desarrollando ni que realice una planificación de lo que ha de suceder. Los roles de los participantes son difusos, ya que los vendedores se pueden convertir en clientes, y viceversa, sin indicación externa.

Al contrario que en el modelo de la catedral, en el bazar hay entregas tempranas del software (*release early*). En el caso de YafRay, como se ha dicho, se liberó desde prácticamente el primer momento. Esto suele motivar que aparezcan rápidamente personas que tenían el mismo problema (en el caso de YafRay, que querían desarrollar un *raytracer*) o que puedan estar interesadas en la solución (que necesitan lo que YafRay les puede ofrecer).

Las pruebas son, como se ha comentado, una de las labores más pesadas en el desarrollo de software. Afortunadamente son un proceso altamente paralelizable. La *release early* de YafRay permitió que muchas personas se pusieran a probar el software simultáneamente, llevando así a cabo la fase de pruebas. Su uso aumentó drásticamente cuando Andrea Carbone realizó la primera versión de Yable, un exportador de escenas desde Blender programado en Python. De este modo Andrea, usuario de Blender, se convirtió en desarrollador de parte del proyecto dentro del bazar YafRay.

Este tipo de relación en que todo el mundo puede aportar algo es altamente productiva. Los usuarios encuentran motivación en encontrar, notificar y corregir errores, ya que saben que su petición va a ser atendida casi inmediatamente. Además existe cierta satisfacción personal por haber aportado algo, algo así como un "yo también he puesto mi granito de arena" de cara al reconocimiento de la comunidad. De hecho, algunas comunidades se apoyan en una *meritocracia*, es decir, las decisiones más importantes las toman las personas que más aportan a la comunidad.

El modelo del bazar, con frecuentes entregas del software, puede asustar a usuarios que buscan la estabilidad. Algunos proyectos, como Blender, mantienen ramas diferentes de desarrollo. Una más estable, en la que las nuevas funcionalidades sólo se añaden después de muchas pruebas, asegurándose que no repercuten en la estabilidad del producto final. Y otras experimentales como Tuhopoo, *the evil tree* o la reciente rama ***Orange***, creada específicamente durante el desarrollo del corto ***Elephant Dreams***.

Pero el bazar no es solo aplicable a los desarrolladores. En ocasiones la vorágine del bazar intimida a nuevos usuarios, lo cuales pueden llegar a encontrar bruscos algunos comportamientos de los usuarios más antiguos. Este recelo de los veteranos de la comunidad suele ser debido a la repetición sistemática de los mismos errores por parte de los nuevos miembros.

Como ejemplo de esto, en la tristemente extinta comunidad de usuarios de Blender hispanos, **Nicodigital**, los nuevos usuarios realizaban las mismas preguntas de novato una y otra vez. Los usuarios más veteranos se cansaban de repetir las mismas respuestas continuamente. Se llegó a hacer famosa la frase *¡¡usa el buscador!!*, en relación al poco uso que daban los novatos al buscador del foro para encontrar posibles respuestas a sus preguntas. Con el doble fin de evitar preguntas innecesarias por parte de los nuevos y motivar a los veteranos a contestar, el administrador del foro (Nicolás Morenas, alias *Caronte*) creó un sistema de créditos. Los créditos se obtenían, entre otras formas, por visitar la web y contestar preguntas; mientras que se perdían por realizarlas. A los pocos meses se creo toda una sociedad basada en el intercambio de créditos. Es una lástima no haber podido seguir la evolución de este experimento, ya que la imposibilidad de mantenerse económicamente, junto con una serie de desavenencias, precipitaron el cierre de la comunidad.

A pesar de la buenas intenciones en el modelo del bazar, las decisiones tienen que estar controladas. Raymond supone que todo proyecto de software libre ha de contar con un dictador benevolente, una especie de líder que generalmente coincide con el fundador del proyecto, para guiarlo reservándose siempre la última palabra en la toma de decisiones. Sobre el papel, esa persona ha de saber motivar y coordinar un proyecto, entender a los usuarios y desarrolladores, buscar consensos e integrar a todo aquel que pueda aportar algo al proyecto. Pero la realidad es que esa persona, como queda dicho, suele ser el propulsor original de la idea. Por tanto es un desarrollador al que le apasiona producir software pero que, habitualmente, se preocupa poco por los usuarios y las relaciones dentro de la comunidad. Este tipo de desarrolladores suele ver como una perdida de tiempo las labores de *ingeniería social* organizativa.

YafRay: de la catedral al bazar organizado.

YafRay ha seguido desde un principio el modelo de procesos en el software libre. Surge como un proyecto personal y la persona que lanza el desarrollo es la que realiza las labores de dictador, considerado por todos benevolente porque si no, nos pegará...

Inicialmente, en la primera etapa de liberación del código (versión 0.0.1), la página web de YafRay era un simple documento *html* de fondo blanco con letras negras. Contenía una pequeña explicación del proyecto, un par de imágenes y un enlace al código. Si YafRay hubiese sido una empresa, esta presentación tan pobre habría desencadenado un **suicidio colectivo** del departamento de marketing. Desafortunadamente, YafRay no pudo hacer este favor a la humanidad ya que era un proyecto de un estudiante de informática. Aprovecho la ocasión para saludar a los chicos de marketing de mi empresa.

A pesar de esta puesta en escena tan austera, el proyecto tomó gran interés por parte de los usuarios de Blender, que por aquel entonces no contaba con un motor de *render* de calidad. Este interés obligó a mejorar aquella web, incluyendo documentación extra y un foro para que los usuarios pudieran solventar sus dudas. Deberíamos reflexionar cómo una buena idea, en el momento adecuado, crece por si misma sin necesidad de envolverla en una nube de *bullshit* dentro de extensas presentaciones de *Powerpoint*.

Ya hemos dado el primer paso de la catedral al bazar casi sin darnos cuenta. La necesidad de crear un foro de usuarios surge de la imposibilidad de Jandro para contestar a todos los correos de los usuarios, que emocionados con el nuevo *raytracer* no paraban de hacer preguntas. El arquitecto de la catedral pierde más tiempo en resolver dudas sobre el proyecto que en su desarrollo. La solución está clara, permitir que la comunidad se comunique para que los usuarios más experimentados puedan ayudar a los nuevos.

De manera casi simultánea al anuncio del proyecto, el holandés Alfredo de Greef comenzó a aportar código, convirtiéndose en uno de los grandes impulsores del proyecto haciéndose responsable de la intergración con Blender. Otros vinieron después, casi todas fueron pequeñas pero importantes aportaciones. En poco más de un año YafRay contaba con una comunidad de usuarios grande, teniendo en cuenta el reducido ámbito del proyecto.

En 2004, durante las jornadas **BoingBoingBlend**, tuve la oportunidad de realizar una ponencia similar a esta. Entonces la situación de YafRay era bastante delicada. El proyecto estaba en crisis. Alejandro no tenía tiempo para dedicar al desarrollo, Alfredo estaba volcado en otros proyectos y no aparecían nuevos programadores. Por otra parte, la poca documentación que existía estaba desfasada y la web lleva años sin cambiar de aspecto. Mis conclusiones de entonces eran bastante pesimistas. El proyecto estaba en una situación de interbloqueo. Las pocas personas interesadas en colaborar, bien desarrollando código o documentando, no sabían como hacerlo porque la poca documentación existente era de mala calidad. Toda la información estaba en la cabeza de los desarrolladores, que precisamente carecían de tiempo para documentar. La pescadilla que se muerde la cola.

El código no estaba libre de problemas. El diseño inicial, pensado para un pequeño *raytracer* ya no soportaba más parches para introducir las nuevas funcionalidades que los usuarios requerían. Eso llevó a Alejandro a pensar en empezar un nuevo diseño desde cero, mejor estructurado y con algoritmos de *raytracer* más avanzados. Para un programador interesado en colaborar era muy complicado ampliar el viejo código y el nuevo no era más que un esbozo.

Con este panorama, las pocas personas que intentaban ayudar, al poco tiempo escapaban del desastre organizativo que era el proyecto. Necesitábamos más compromiso y para ello optamos por un arma infalible: **la vergüenza pública**. Durante aquellas jornadas de Barcelona, en mitad de la ponencia pedimos a ciertas personas que colaborasen con el proyecto y que se comprometieran allí mismo, cara a cara con otros miembros de la comunidad. La jugada no salió mal: Apenas recibimos amenazas de muerte y conseguimos que Javier Galán diseñase una nueva imágen para YafRay y su web, mucho más profesional.

En mi opinión aquello fue clave para el estado actual de YafRay. Dos años después, la documentación está creciendo gracias al ingente trabajo de Alvaro Luna. **Mathias Wein** ha tomado las riendas del proyecto, introduciendo grandes mejoras al código original, mientras el nuevo rediseño (llamado *Fry*) va madurando. Quiero agradecer desde aquí a estas personas y a tantas otras que con su trabajo, sus mensajes en los foros y sus trabajos artísticos de calidad profesional están consiguiendo que YafRay crezca cada día más.

Desarrollo libre para el software libre

Hemos hablado de la cooperación de las comunidades en el desarrollo de proyectos de software libre, pero ¿Cómo se comunican sus miembros? ¿Qué herramientas se utilizan para coordinar los esfuerzos? Afortunadamente la comunidad de software libre se retroalimenta y existen multitud de herramientas libres útiles para el desarrollo de software y la cooperación entre personas de todo el mundo.

Los foros y las listas de correo son el epicentro de las comunidades. De hecho son la parte visible de la comunidad. Ambas herramientas permiten la comunicación y las dos comparten la idea de facilitar el envío de mensajes a todos los miembros. Su diferencia fundamental el sentido en el que viaja la información. En las listas de correo la información llega pasivamente a los usuarios, los mensajes enviados a la lista llegan a toda la comunidad. Sin embargo en los foros son los usuarios los que se "acercan" a la información. Por esto, las listas se deberían utilizar para intercambiar información útil para todos sus destinatarios. Un ejemplo clásico es una lista para los desarrolladores, donde cada miembro envía un registro de sus modificaciones.

En los foros, sin embargo, el usuario es el que decide leer o no un mensaje. Se dividen en secciones y cada usuario lee sólo la que le interesa. La información está ahí, sólo hay que ir a buscarla. Son una herramienta útil tanto para usuarios como para desarrolladores. No sólo se utilizan para

resolver dudas sobre el uso de un software, sino que sirven para poner en contacto a los desarrolladores y los usuarios.

Una de las herramientas que está invadiendo la escena del software libre es el *Wiki*. Del hawaiano *wiki wiki*, "rápido", es una forma de sitio web en donde se acepta que usuarios creen, editen, borren o modifiquen el contenido de una página web, de una forma interactiva, fácil y rápida. Dichas facilidades lo convierten una herramienta efectiva para la escritura colaborativa. En el proyecto YafRay, la instalación de un wiki (una wiki para algunos (algunas para algunos/as)) le ha permitido salir de la penosa situación documental que se encontraba durante el último año. Se está redondeando una documentación completa y actualizada. Además la wiki ha permitido que los usuarios pudieran traducir por si mismos la documentación a otros idiomas, como el español, portugués o chino.

En cuanto al desarrollo en si, hay que decir que YafRay empezó siendo desarrollado con herramientas libres, desde el compilador GNU hasta el V*im* con el que Jandro tecleó las primeras lineas. Actualmente el único software propietario que se utiliza es el compilador Visual C Toolkit (la versión gratuita del compilador de Microsoft para Windows,). Aunque inicialmente se utilizó el **Cygwin** como herramienta de compilación para Windows, la idea se abandono ya que la integración con Blender, usando YafRay como *plugin*, requería el uso del compilador de los de Redmond.

En el momento en el que más de una persona aportaba código al proyecto se hizo necesario el uso de una herramienta para el control de versiones, CVS en el caso de YafRay. El **CVS** (*Concurrent Versions System*), implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en la desarrollo de un proyecto (de programa) y permite que distintos programadores colaboren. La mayor parte de los proyectos de software libre utilizan esta herramienta o la más avanzada ***subversion***.

Para terminar este pequeño repaso por las herramientas de desarrollo quiero hacer una mención a las utilidades de tipo *Bugtracker*, literalmente "rastreador de errores". Se trata de una especie de foro (en muchas ocasiones se utiliza como tal) en el que los usuarios dejan constancia de los fallos que van encontrando en el software, permitiendo en muchos casos añadir sugerencias sobre nuevas funcionalidades (*feature requests*). El *Bugtracker* permite asignar programadores al problema, que serán los encargados de investigar las causas del error y subsanarlo. Es una herramienta muy potente y que recomiendo a todos aquellos proyectos que empiecen a tomar cierta relevancia.

Conclusiones

El ámbito de aplicación y desarrollo del software propietario suele (o debe) estar definido de antemano. Se puede realizar una predicción, más o menos acertada, sobre la envergadura del proyecto así como del número y perfil de los usuarios finales.

Pero el software libre es totalmente impredecible. Un proyecto inicialmente pequeño, comenzado desde cero por una única persona puede volverse muy grande en muy poco tiempo.

El desarrollador que empezó por diversión empieza a ver que los usuarios demandan su ayuda y piden nuevas funcionalidades al software. Aparecen los primeros *bugs* que hay que resolver, lo que genera nuevas peticiones de los usuarios. Todo empieza a dejar de ser divertido. Usuarios curiosos inundan la cuenta de correo del programador. Aparecen nuevas personas intentando colaborar. En este caos organizativo el desarrollador debe asumir que su proyecto es de interés para la comunidad, utilizando parte de su tiempo en la organización de la comunidad que se ha formado alrededor de su software. Muchos lo verán como una perdida de tiempo, pero es necesario, ya que una buena planificación a tiempo favorecerá un desarrollo más fluido del proyecto.

Los desarrolladores han de tener en cuenta que afortunadamente no están sólos, ya que existen una amplia gama de herramientas libres que permiten crear comunidades organizadas. Sabiendo el potencial de estas herramientas se podrá poner en las manos de los usuarios la oportunidad de colaborar. Conociendo las alternativas y eligiendo los instrumentos adecuados se puede formar una comunidad que se “automantenga”, haciendo que sean los usuarios los que contesten sus dudas y escriban la documentación.

Uno de los principales errores de YafRay fue descuidar estos aspectos y ha costado mucho levantar el vuelo. Si tú, amigo lector, estás empezando a desarrollar software libre ten en cuenta que deberás dedicar parte de tu tiempo al cuidado de las personas que se agrupan alrededor de tu proyecto. Si por el contrario lo tuyo no es programar y el menú del vídeo es todo un misterio para ti, no te preocupes. Hay mucha labor que hacer en el proyecto de tu software favorito. Cualquier pequeña aportación, como ayudar a los usuarios más inexpertos, puede hacer mejorar el software.

No quiero terminar sin agradecer a Alejandro Conty la oportunidad de participar en un proyecto como YafRay. Mil gracias también a Alfredo de Greef, Javier Galán, Mathias Wein, Alvaro Luna y a toda la comunidad hispana e internacional de usuarios de YafRay, con especial cariño a los de Blender...

Llegados a este punto sólo queda contestar a una pregunta: ¿Cuantos robles roería un roedor, si los roedores royesen robles? Bien... un roedor no roería robles, ya que los roedores no roen robles, pero si un roedor pudiera roer y royera alguna cantidad de robles, ¿cuántos robles roería un roedor? Aunque un roedor pudiera roer robles y aunque un roedor royera robles, ¿debe un roedor roer robles? Un roedor debería roer si un roedor pudiera roer robles, siempre que el roedor royera robles.

Oh, ¡**callate**!

Referencias

- Gregorio Robles. "*Ingeniería del Software Libre. Una visión alternativa a la ingeniería del software tradicional*". 2002.
- Jesús González Barahona, Joaquin Seoane, Gregorio Robles. "I*ntroducción al software libre*". 2003
- Eric S. Raymond. "*La catedral y el bazar*". 1997
- Martine Aalbers, "*Motivation for participating in an online open source software community*". 2004

- `http://pulsar.unizar.es/gluz/manual-sl/c35.html`
- `http://es.wikipedia.org`

¿La Retrocomputación está de moda?

Javier Belanche Alonso
xbelanch@gmail.com ::



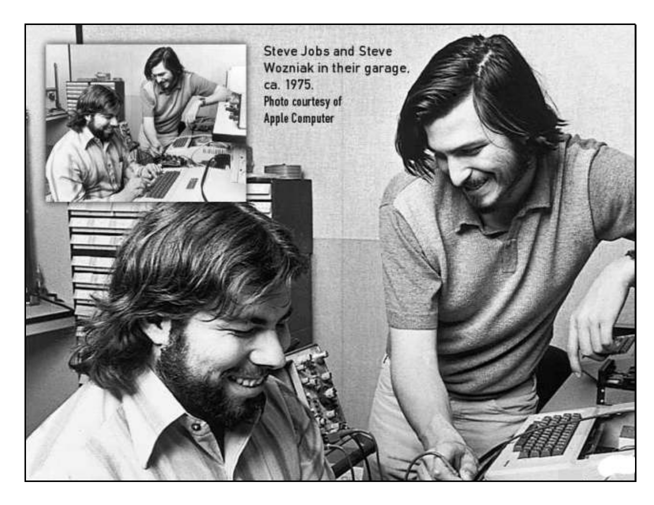
¿La retrocomputación está de moda? Es evidente que sí. El sentimiento de nostalgia por la tecnología del ordenador personal, consolas y máquinas recreativas (no puedo encontrar un nombre mejor) de la época comprendida entre mediados de los 70 y 80, ha ido tomando mayor fuerza y visibilidad en el momento actual.

Esta charla tratará de describir las diferentes manifestaciones de la retrocomputación y, en particular, de la necesidad para muchos de volver a repetir el embargo emocional que representa jugar, programar, construir y coleccionar la informática de 8 bits.

Un poco de historia

El punto de partida es la aparición del primer videojuego comercial en Estados Unidos en 1974. Pong, idea y diseño de Nolan Bushnell y formalizado por el ingeniero de Al Alcorn, inaugura el festival del ocio informático popular como la industria que, a excepción de un tiempo de crisis a mediados de la década de los 80, en la actualidad factura un volumen de ingresos superior a la del cine.

De la primera máquina recreativa surge la idea, poco más tarde, de Nolan Bushnell de trasladar la experiencia de las máquinas recreativas al entorno doméstico; reiventa un proyecto tan fallido como visionario, la consola Magnavox Odissey. La edición "home" del Pong fue un éxito de ventas de las navidades del 75. Es la época de confrontación entre Atari y Colecovision por dominar el mercado de máquinas recreativas y consolas domésticas. Frente a la producción masiva de Colecovisión, la Atari de Nolan Bushnell determinará, mediante el acto de la innovación, un tercer paso hacia la definición contemporánea de la consola: la invención del "cartucho" o la posibilidad de intercambiar juegos en la misma consola. Seguirán años de creatividad e innovación en cuanto al diseño y producción de juegos para dos mundos que conviven inevitablemente, el de las máquina recreativas, superior en calidad gráfica al segundo, y el de las consolas domésticas. Space Invaders, PacMan, Donkey Kong, Asteroids



se convertirán rápidamente en la primera colección de clásicos de la historia de los videojuegos. No sorprende, por lo tanto, que seran objeto de variaciones, con mejor o peor gusto hasta nuestros días y que tendrán representación en soportes y medios tan diferentes como singulares.

La segunda parte de esta historia la ocupa el ordenador doméstico. Tres años después del éxito de Pong, Steve Jobs y Steve Wozniak presentan en 1977 el Apple][, con el procesador 6502 de la casa MOS y que tendrá un enorme eco comercial, en parte por su cualidad técnica y en gran medida por la aplicación de ofimática Visicalc, la primera hoja de cálculo de la historia.

Pero el principal interés que nos trae aquí la obra de Wozniak y Jobs es por un motivo clave para la evolución del videojuego: gracias al intérprete de BASIC, los usuarios podían desarrollar y programar sus propios juegos, almacenarlos y, finalmente compartirlos o venderlos. De nuevo, como fue en el caso de Atari, la empresa Apple descubre una doble industria (la del software y la del hardware) que, en poco años, se inundará de un largo número de ordenadores domésticos, de los que destacamos la producción del Commodore 64, Sinclair ZXSpectrum 48k, MSX y CPC 464 de AMSTRAD. Durante los primeros cinco años de los 80, la explosión de creación de



videojuegos para los modelos antes mencionados se verá reforzada por el desarrollo, entre profesional y el amateur, de muchos jóvenes que aprenden a programar ensamblador para dos micros, el z80 y el 6502, en sus habitaciones, aspirando a poder vivir de sus propios trabajos. No resulta extraño que, desde la perspectiva actual, se considera ese momento como la edad de oro del software español. Edad que vió su punto más álgido con la obra maestra del programador Paco Menéndez, "La abadía del Crimen".

Punto final a esta historia lo pondrá Nintendo o, por qué no, del encuentro entre Yamauchi y Shigeru Miyamoto. En 1985 aparece la consola NES de Nintendo y su juego estrella, Super Mario Bros, y que culmina dos años más tarde con The Legend of Zelda.

Intenciones

He clasificado el contenido de esta ponencia en función de unos pocos conceptos que engloban la gran totalidad de proyectos que han girado alrededor de la informática de 8 bits.

- Coleccionismo
- LowProgramming/HardWiring
- Emulación/Recreación
- Reinterpretación/Reinvención

La mayor parte de manifestaciones o acciones "retro" puede referirse a estos cuatro apartados básicos, y que, si bien se pueden tomar por apartados independientes, la estrecha relación entre ellas se hará evidente a lo largo de la explicación de este texto. Empezaremos con el más pasivo de los cuatro, coleccionar, y acabaremos por el que más se aleja de la retroinformática y sí se aproxima al medio de la práctica del arte actual.

Coleccionismo

Coleccionar, reunir o agrupar objetos de una misma índole o tema, que despiertan un sentimiento entre nostalgia, pasión y deseo, es probablemente la categoría más pasiva de las cuatro. La acumulación de ordenadores domésticos de 8 bits, consolas, calculadoras no es diferente de la de coleccionar sellos, sin ocuparnos ahora si hay sospechas de un futuro fórum retroinformático. En España contamos con algunos de las colecciones privadas de retroinformática más importante a escala internacional.

Sin duda, el componente de interés del coleccionismo es el encuentro con lo diferente, la excepción o la singularidad y, por lo tanto, con el objeto más preciado. Quiero destacar varias piezas, explicar su excepcionalidad como la magia y encanto que han suscitado desde su desaparición en el mercado informático. Es probable que la elección no sea la correcta, la mejor o la más representativa, pero cada vez que aparecen en subasta, los precios finales rebasan habitualmente los seiscientos euros.



En primer lugar, Jupiter Ace de Cantab, ordenador británico que, literalmente, nació muerto a principios de los 80. Creado por Richard Altwasser y Steven Vickers, parte escindida del equipo original del desarrollo del ZX Spectrum, el Jupiter Ace heredaba del ZX81 su fisonomía, pero de superficie blanca. Micro Z80 a 3.25 MHz con 3k de RAM ampliables a un máximo de 48 y una resolución gráfica de 24x32 caracteres en blanco y negro, el Jupiter Ace se fundamentó en la estandarización de sus componentes digitales. El Ace, a diferencia de las ULA (Unidad Aritmético Lógica) de la línea Sinclair, se diseñó bajo el paraguas de la lógica transistor-transistor y, por lo tanto, con chips de serie. Quizá, su peculiaridad mayor fue el intérprete de Forth en lugar del habitual BASIC. El Ace contenía una ROM de 8k con el SO y el intérprete de Forth. Aunque la velocidad de ejecución era superior al resto de intérpretes de BASIC del momento, el Jupiter Ace se encontraba en desventaja frente a otras máquinas debido a su insuficiente capacidad gráfica como de sonido, limitando las ventas del Ace hasta el punto de su desaparición tres años mas tarde.



En segundo lugar, el Sam Coupé, también británico, hace su aparición en 1989, cuando el mercado de los ordenadores domésticos de 8 bits ha entrado en caída libre. La empresa



responsable fue Miles Gordon Technology. Al igual que el Jupiter Ace, el Sam Coupé tenía el z80 de Zilog por microprocesador, a 6Mhz de reloj, 256Kb de RAM ampliables a 512kb. Una característica que luego se convirtió en motivo de confusión fue la posibilidad de ejecutar juegos de la gama Sinclair gracias al chip ASIC, compatible con el ULA del ZX Spectrum 48k. Confusión que llevó a entenderse el Sam Coupé como un clónico ampliado del ZX Spectrum. Lamentablemente, después de tres cambios de propietarios, la producción del Sam Coupé finaliza en 1992, tras 16000 unidades vendidas y con el imparable mercado del PC compatible como fondo.

Finalmente, el Rainbow-I fue un ordenador de 8 bits que apareció en 1989 en la localidad de Manchester, bajo el diseño de los hermanos Ian y



Stephen Smith, dos jóvenes apasionados de la electrónica digital, la informática, la música de "Madcheter", la literatura y arte decadente inglés del siglo XIX y las drogas. El Rainbow-I nunca superó las 100 unidades: la caja era de madera de cerezo, con grabados de ornamentación en gran parte de su superficie, se vendía a petición personal o se regalaba a colegas. Más en la línea del romanticismo del "homebrew computer" que de la producción comercial, el Rainbow-I tuvo un cierto eco en revistas especializadas debido al uso de dos micros z80 en paralelo, mejorando considerablemente la profundidad de color. Un año más tarde, Ian Smith se suicidó destruyendo toda la información y el poco stock disponible de los Rainbow-I. De muy vez en cuando, aparece como artículo de subasta de ebay, en particular de la región del Reino Unido, algun prototipo del Rainbow-I, superando los 2500 euros.

Emulación/Recreación

Hablar de la vida de estas máquinas nos lleva a la segunda categoría, la emulación/recreación. La emulación es la solución pobre de una experiencia real, pero si sumamos la recreación, la sensación es mayor y más próxima al sentimiento original. La retro-emulación es un extraño virus: se extiende con suma facilidad a todas los dispositivos digitales actuales, desde relojes hasta las consolas de última generación, pasando por pdas. El interés de la emulación es doble: la satisfacción que produce el reproducir el comportamiento de una vieja máquina, una cierta necrofilia digital y, una línea quizá menos habitual, pero no por ello menos atractiva y enriquecidora, la programación de emuladores de máquinas de 8 bits.

El principio que subyace la emulación en el campo de la informática es la compatibilidad. Que el software de un primer ordenador sea compatible con un segundo es posible si, gracias si está disponible el código fuente y podamos compilar el binario en la segunda máquina. Tarea esta última difícil si entre las dos máquina la arquitectura és diferente. Un camino alternativo es la "ilusión" de poder "emular" el comportamiento del primer ordenador en el segundo. De esta manera podríamos ejecutar el software del primer ordenador sin dificultad en el segundo.

Los primeros emuladores de máquinas viejas de 8 bits aparecen a principios de los 90, con la desventaja de la todavía precaria velocidad del micro para ejecutar con suavidad el sistema emulado, como un emulador del Commodore 64 para Amiga.

Con el rápido incremento de la velocidad de los microprocesadores para los PC compatible, la aparición a mediados de los 90 de una "emuscene" se cristalizó en los primeros emuladores de la serie de máquinas de 8 bits de principios de los 80, Sinclair ZX Spectrum, MSX, Commodore 64, Amiga e incluso Apple][, pero también las primeras consolas como la NES, la Super NES o la Sega Master System. La ventaja en cuanto a implementación y programación de estos emuladores es que todas estas máquinas guardan similitudes técnicas y de diseño muy parecidas; muchos compartían micros de 8 bits, Interl 8080, Motorola 6809, Zilog Z80 o MOS 6502. Las diferencias venían por los dispositivos externos (teclado, cintas o disco de almacenamiento) o el chip gráfico (la ULA famosa del ZX Spectrum) o del sonido.

Contrariamente a lo que podamos pensar, la documentación alrededor del diseño y programación de emuladores es bastante limitada y dispersa si la comparamos con el volumen de comunidades, proyectos y usuarios interesados en la emulación desde su arista más lúdica. En general, la motivación que hay detrás de la programación de emuladores es variada: aprender más sobre computación y la arquitectura de los ordenadores y

consolas de 8 bits, saber cómo funcionaban los videojuegos de antes y qué había detrás de ellos -pregunta muy razonable cuando éramos niños- y, desde un lado más humano, preservar la memoria de nuestra infancia.

Acorde al movimiento de la 'emuscene' y del sentimiento de preservación del pasado de 8 bits, el 5 de febrero de 1997 Nicola Salmoria publica la versión 0.1 de MAME, acrónimo de Multi Arcade Machine Emulator, siguiendo una filosofía estricta de la emulación a nivel de registros e instrucciones, contrariamente a la técnicas High Level de Emulación. La velocidad de la emulación no es tan importante como el grado de realismo de la emulación, por lo que MAME dependerá siempre de velocidades de reloj enormemente elevadas para una emulación fluida.

MAME representa en la actualidad un gigante centro de gravedad en lo que se refiere al universo de la emulación de videojuegos: el número de juegos que alcanza emular está en los 10000, aunque también se sabe que alrededor de 600 se encuentran fuera de los límites de su emulación por MAME. Al mismo tiempo, MAME representa un quebradero legal: en cuanto a emulador, la pertenencia y distribución de MAME es legal, su venta es ilegal. Respecto las ROM, la mayoría están sujetas en la actualidad al copyright de la empresa o autores propietarios y, por lo tanto, imposible su venta y distribución. Esta situación se vuelve surrealista en función de la situación legal de cada país, en el hecho de que las ROMS dejaron de ser objeto de mercado hace tiempo o que, rozando el ridículo, algunas empresas propietarias de las ROMs dejaron de existir. Estas ROMS pasan a ser consideradas huérfanas, pero mantienen paradójicamente su copyright, sin que por ello beneficie económicamente a nadie. Lawrence Lessig, padre del Creative Commons, debate y denucia en la actualidad la irregularidad del copyright de los juegos huérfanos y defiende el traspaso al dominio público.

He mencionado más arriba la técnica de emulación High Level, en contraposición de la filosofía seguida por los desarrolladores de MAME. Dos años después de la primera versión pública de MAME, se anuncia un nuevo emulador, UltraHLE, que difiere en cuanto a metodología de emulación y por tanto, el enfoque conceptual de emulación. A diferencia de MAME, que representa la técnica LLE (Low Level Emulation) y concentra sus esfuerzos en emular a bajo nivel todo los componentes del sistema emulado, con especial atención en la CPU, UltraHLE intenta adivinar que necesita el juego en cada momento y emula la petición con gran rapidez. De hecho, UltraHLE se hizo enormemente popular en muy poco tiempo. Fue el primer emulador con éxito de la Nintendo 64, de juegos imposibles de emular como Super mario 64 y Zelda: Ocarina of the Time.

UltraHLE murió de éxito repentino. Pocas horas después de su publicación, los desarrolladores abandonaron el proyecto, aunque poco después, RealityMan, uno de los desarrolladores, volvió a la carga para sucumbir debido a las sugerencias de Nintendo para olvidar el proyecto, incluso la emulación.

Recreación

La recreación es bicéfala: busca obtener una experiencia más rica al dotar a la emulación de un soporte físico específico y próximo al original. Las retroMacas ilustran a la perfección esta idea. La lástima es que esta recreación se limita a la habitación del retrogeek. La recreación perfecta, casi como siempre, es colectiva y en este sentido la experiencia original se debería completar en una sala llena de humo de tabaco negro, ceniceros quemados y llenos de nicotina, el estar a punto de conseguir una bandera 10 en el Galaxian mientras una multitud anónima y silenciosa admira tus movimientos y, cómo no, un jubilado con mono azul y riñonera llena de monedas de 25 de las viejas pesetas.

La aparición de MAME, su capacidad técnica para emular fielmente los videjuegos clásicos, lo convirtieron en el software ideal para la recreación de una máquina recreativa, o "maca". La construcción/obtención/compra de una "maca" se ha convertido en sí mismo en fenómeno unido a la experiencia de la retro emulación. Podemos comprarla de segunda mano, readaptarla, recuperarla del cementerio de recreativas; podemos construirla desde cero (from scratch) y ser nosotros quienes definamos el diseño, entre el clásico y el "cocktail", o estilo mesilla. El mercado de "kits de construcción" de macas es cada vez mayor; se venden accesorios como los monederos clásicos, las "arcade controls" y tarjetas gráficas de frecuencias de refresco inferiores a los 60Hz, J-pac...





Hago un alto. He de hablar de la GP2X, sucesora de la malograda GP32. Tras una tortuosa gestación y sin llegar al año de vida en el mercado virtual (la consola oficialmente sólo se distribuye a través de internet), la GP2X ha sabido encontrar un público fiel y adepto a la retroinformática. Gran parte de la atención recibida se debe a su orientación de desarrollo libre de videojuegos, aplicaciones y emuladores. Sin pasar del límite de los 170 euros, la GP2X contiene un firmware -finalmente- libre (basado en GNU/Linux) sobre un procesador ARM9 de doble núcleo, velocidad reloj de 200Mhz, almacenamiento en tarjetas SD, 64mb de RAM y resolución de pantalla de 320x240 (QVGA). Dispone de un SDK oficial igualmente bajo licencia GNU, aunque la mayoría de programadores prefieren usar SDL nativas de la GP2X.

Destacar la importancia de la escena española concentrada en la comunidad gp32spain, liderada por Franxis.

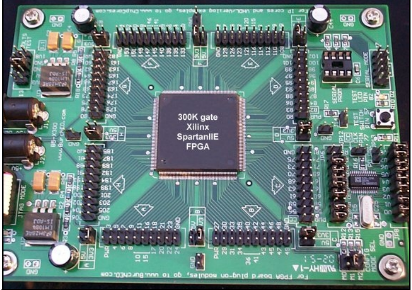
Al basar su sistema operativo en GNU/Linux, no ha sido raro que la consola se beneficiera rápidamente de las comunidades. Destacar que es la primera consola de mano que tiene un intéprete de Python (Pygame) y Ruby para el desarrollo rápido de videojuegos y aplicaciones.

Hardware Emulation

La máxima expresión de la emulación/recreación corresponde al uso de chips FPGA que traspasan la frontera misma del la emulación por software. Identificamos la clonación perfecta mediante la emulación por hardware al uso de la recreación de máquinas y consolas de 8 bits antiguas. Los chips de Altera y Xilinx, el lenguaje vhdl y un enorme conocimiento de la máquina a emular (y tiempo, miucho tiempo) se suman para aparecer en forma de viejas máquinas recreativas y antiguos ordenadores domésticos. Quizá, del conjunto de proyectos basados en FPGA orientados a la retro experiencia, señalar sin duda el de Jeri Ellsworth, autodidacta en la programación de chips FPGA, el C-One.



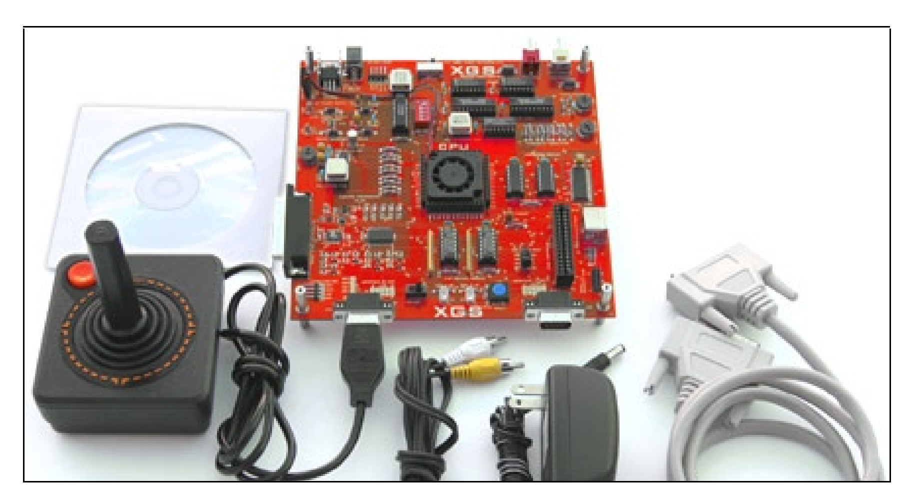
C-One representa una iniciativa de Jeri Ellsworth por mejorar el Commodore 64 original, pero que finalmente derivó en una placa base que permitiera la clonación de cualquier máquina de 8 bits. C-One es una suerte de multi-hardware (en lugar de multiplataforma) que, mediante el uso de FPGA y una expansión modular de la CPU (que por defecto incorpora el 65C186 del Commodore 64) se logra implementar la emulación por hardware de cualquier máquina de 8 bits y que, por su naturaleza abierta y modular, permite nuevos diseños personalizados.



Hardwiring/Low programming

El año pasado, André LaMothe, autor especializado en la publicación de libros sobre diseño, creación y programación de videojuegos (cuenta con casi una veintena de libros), da un giro radical de contenidos y celebra una vuelta radical al pasado: crear videojuegos desde una perspectiva "low-level": electrónica, diseño digital, programación en ensamblador, conocimiento del conjunto de instrucciones del micro. Esta vez el libro no es un simple recetario de algoritmos, ejemplos y código para nuestro enésimo proyecto de videjuego fallido en DirectX. En su lugar, el libro es

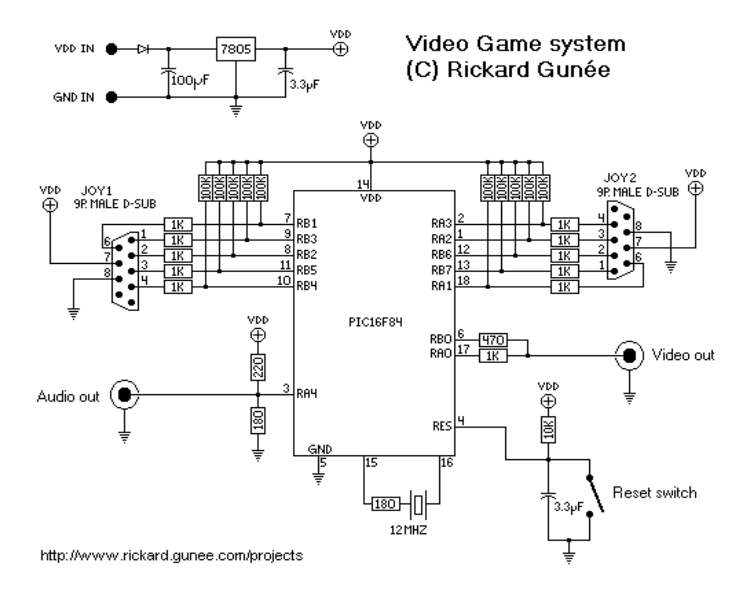
una mezcla de curso de electrónica y justificación de la elección del diseño de la Xgamestation, una consola basada en el microcontrolador SX52 y de la que los usarios pueden aprender a programarla -en ensamblador, por supuesto- o ampliar.



¿Es la Xgamestation un caso aislado? Dentro del libro, Andre LaMothe explica el desarrollo de una edición mínima del Xgamestation, el Pico Edition. Un protoboard, el SX52 y un set de resistencias y algún condesador proporciona una infraestructura mínima en el que podemos jugar a nivel de montaje (hardwiring) y de programación del SX52 (low-programming). El Pico Edition no es novedad y André LaMothe debe su iniciativa a un anterior proyecto personal de un estudiante sueco de electrónica, Rickard Gunee, al utilizar microcontroladores de la familia PIC y SX para diseñar sistemas de videojuegos amateur. La sensibilidad "homebrew console" es evidente y trascienden el simple acto recreativo al puro cacharrismo "Make" (revista actual dedicada al hardware y tecnología amateur): nuevos proyectos de "homebrew computers" (incluso con intérpretes de BASIC!) y "homebrew games".

"Homebrew games" representa un paso final que nace desde el interés de la emulación de retro videojuegos, pasando por el estudio de cómo estos juegos se programaron entonces y finalmente programando un juego que probaremos en nuestro emulador favorito y que, una vez conseguido, desearemos transformar el juego en un cartucho.

La escena de los "homebrew games" es todavía muy incipiente. Programar en ensamblador en estos días parece un acto de excentricidad y de frikismo. Añadir la dificultad que ello comporta en cuanto al aprendizaje como el de un suficiente control para poder desplegar sprites y la



animación de éstos por la pantalla. El último reto, superado lo anterior, pasa por la fabricación del cartucho. De nuevo el puente entre el low-programming al hardwiring o viceversa. Habitualmente, los pocos creadores de "homebrew games" tienen dos opciones: cambiar la ROM de un viejo cartucho y reemplazarla por la nueva ROM o, el más difícil todavía, crear tu propio circuito impreso (PCB) y el packaging del cartucho. Un ejemplo magistral de lo que hablo son las tiradas limitadas del juego "I, Cyborg", de George Peloni, para la mítica consola "Vectrex".

La retroinformática y el arte actual: la deconstrucción como reinvención de los clásicos

He querido dedicar el último apartado de esta charla a las relaciones que, desde el 2000, se han ido estableciendo entre los videojuegos y el arte actual y que, tal como ya dije al principio, es la cara de la otra moneda del sentimiento del retrocoleccionista. Gran parte de la exploración del artista en la iconografía de los videojuegos remite a clásicos de los 70-80 como Arkanaoid, Asteroids, Space Invaders, Pong o Pac-Man. Generalmente, la finalidad de estas intervenciones es desvirtuar el propósito original del juego (matar marcianos, vencer a tu contrincante, ganar a la máquina o tomar todas las pastillas en un lugar oscuro bajo música electrónica.) y presentar un discurso diferente, contrario o inmerso en la ironía.

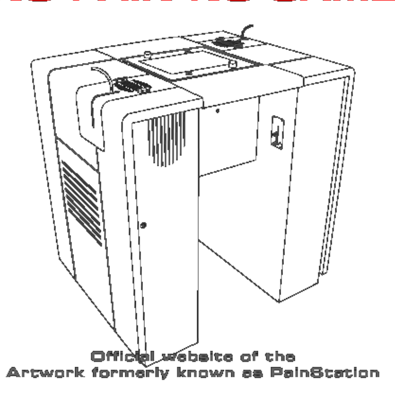
Si un estudio detallado de la relación entre el arte actual y los videojuegos escapa a los límites de esta charla, no pierdo la oportunidad de mencionar unos pocos, en particular dos: el proyecto spaceinvaders y la PainStation.

SpaceInvaders es un proyecto personal del frances "invaders" de intervención urbana: la invasión del espacio cristalizada en forma de pequeños mosaicos que, no es casualidad la elección del soporte, reivindican el píxel como unidad de medida gráfica. Sus mosaicos de invaders estan repartidos por toda la geografía del primer mundo, en especial Europa y Estados Unidos.



PainStation es una "maca" especial del tipo mesilla. Los jugadores compiten en el clásico terreno del Pong, a diferencia que, el jugador que pierde un tanto de partida recibe una pequeña descarga eléctrica.

NO PAIN NO GAME



Para saber un poco más

- Forster, Winnie, "Game.Machines, The encyclopedia of Consoles, handhelds & home computers 1972-2005", 2005, Gameplan
- Moya del Barrio, Víctor, "Study of the techniques for emulation programming (by a bored and boring guy)", 2001, UPC
- Kohler, Chris, "Retro Gaming hacks, tips & tools for playing the classics", 2005, O'Reilly
- LaMothe, André, "The Black Art of Video Game Console Design", 2006, SAMS
- Martínez, David "De Super Mario a Lara Croft, la historia oculta de los videojuegos", 2003, Dolmen Editorial



Sección II

Fundamentos

Fotograma de la Película de Animación
"Elephants Dream" realizada con Blender



What's this?

It's simple... Just a *monkey*

Texturas Procedurales

Carlos López Yrigaray
klopes@unizar.es ::

Fundamentos de los conjuntos fractales

Conjuntos conceptualmente simples, como puntos o líneas en un espacio matemático común como el euclídeo, no dejan lugar a dudas sobre sus propiedades características. Un punto, por ejemplo, sabemos que tiene dimensión 0 porque es la intersección de dos rectas que se cortan, e igualmente un segmento de curva tiene dimensión 1 porque, sin ser un punto, puede incluirse en una recta y cumple la exigencia intuitiva de algo continuo.

Condiciones básicas de dimensión

No todos los conjuntos son tan simples, a pesar de poder ser encerrados en *trozos pequeños*, o cualquier, poder generarse sin levantar el lápiz del papel, o por sencillas reglas recursivas. De hecho, la dificultad de la definición del conjunto no tiene que ver con la complejidad de su representación gráfica. Para estos conjuntos en los que las definiciones intuitivas de sus propiedades no parecen encajar, se hace necesaria una definición rigurosa, como la del caso que nos ocupa: el concepto de dimensión.

Para abarcar las posibilidades de la definición de dimensión para un conjunto $X \subset \mathbb{R}^n$, exigiremos cumplirse las siguientes propiedades:

1. dim{p}=0, dim(I)=1, y en general, $\dim(I^n)=n$ para el hipercubo n-dimensional I^n
2. Monotonía: Si $X \subset Y$ entonces $dim(X)<dim(Y)$
3. Estabilidad contable: Si $\{X_j\}$ es una sucesión de $\mathbb{R}^n$ conjuntos cerrados de , entonces

$$dim\left(\coprod_{j=1}^{\infty} X_j\right)=\sup_{j\geq 1} dim(X_j)$$

4. Invariancia: Para una aplicación arbitraria φperteneciente a $\mathbb{R}^n$ cierta subfamilia del conjunto de homeomorfismos de en $\mathbb{R}^n$, se tiene:

$$dim(\varphi(X))=dim(X)$$

Las condiciones (1) y (2) se cumplen fácilmente, mientras que la tercera se puede flexibilizar requiriendo sólamente una cantidad finita de subconjuntos de $\mathbb{R}^n$:

3'. Si $X_1, X_2 \ldots X_m$ son subconjuntos cerrados de $\mathbb{R}^n$, entonces

$$dim\left(\coprod_{j=1}^{m} X_j\right)=\max_{1\leq j\leq n} dim(X_j)$$

La condición (4) implica invariancia topológica del concepto de dimensión. No es trivial ver que las condiciones (1) y (4) no se contradigan, como podría parecer por la construcción que Peano hizo de una aplicación continua de [0,1] en el cuadrado [0,1]x[0,1]. Afortunadamente, esta aplicación no es homeomorfismo por no ser biyectiva.

Dimensiones topológica y de Haufsdorff

Urysohn y otros construyeron una función dimensión que es invariante topológica y toma valores enteros. A esta la llamaremos dimensión topológica, dim_T.

Se basa en la idea de dar el valor n+1 al conjunto del cual otro que tiene dimensión n es borde, inductivamente. En topología general se definen tres tipos de dimensión sobre conjuntos de un espacio métrico contable. Podemos llamar dimensión topológica a cualquiera de ellas, puesto que los valores enteros son coincidentes en las tres.

A finales del siglo XIX, Borel investigó sobre la idea de longitudes, áreas y volúmenes en busca de definición precisa, lo que estimuló a Lebesgue en su teoría de la medida. Carthéodory generalizó este trabajo a la medida *s*-dimensional en $\mathbb{R}^n$ y, finalmente Haufsdorff, viendo que *s* daba sentido a la teoría aún sin ser entero, definió la dimensión que lleva su nombre.

La dimensión de Haufsdorff satisface las condiciones impuestas al principio tomando como familia de aplicaciones de la condición (4) el conjunto de funciones bi-Lipschitzianas. Denotamos a la dimensión de Haufsdorff como dim_H.

Conjuntos fractales

En general, se cumple $dim_T(X) \leq dim_H(X)$. Decimos que un conjunto X de $\mathbb{R}^n$ es *fractal* cuando:

$$dim_T(X) < dim_H(X)$$

También definimos de forma pareja, el *grado fractal*, que nos da la idea de la complejidad gráfica del conjunto:

$$\delta(X) = dim_H(X) - dim_T(X)$$

Esto conlleva, en primer lugar, que conjuntos como puntos aislados o segmentos no son fractales, por ser su dimensión topológica un número entero y, por tanto, igual a la de Haufsdorff.

Debemos, pues, alejar la idea de que los conjuntos fractales son aquellos que lo parecen por ser gráficamente complejos, o los que albergan el concepto de auto-similaridad que, si bien es característico en muchos de estos conjuntos, no es una propiedad exclusiva.

Texturas iterativas

Comoquiera que la idea de *lo fractal* implicaría la realización de infinitos cálculos, lo que podemos conseguir mediante técnicas por ordenador es iterar el proceso de obtención de estos conjuntos para acercarlos al límite de lo distinguible por los sentidos. Esto no sólo nos permitirá representar estos conjuntos de forma satisfactoria, sino que podremos flexibilizar el proceso para conseguir dibujos con menos detalle que el que daría el cálculo completo, si así lo deseamos.

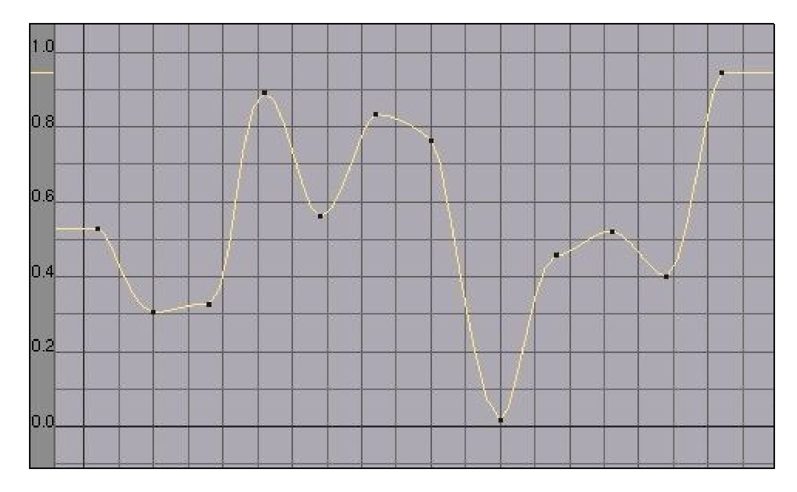
Aquí abandonaremos la palabra fractal y lo que conlleva, para retomar el tema como la generación de gráficos complejos mediante técnicas iterativas.

Las texturas que nos ocupan se basan en variaciones de un proceso de cálculo sencillo, que se aplica a conjuntos de números generados por distintos tipos de *ruidos*. Veamos qué es esto.

Funciones de ruido

Iniciamos la generación de texturas mediante ruido creando una función $\mathbb{R}\rightarrow[0,1]$ base, que no es más que la interpolación de un número (discreto) de puntos aleatorios $P_1, P_2, \dots, P_m$ cuya segunda coordenada está elegida en el rango [0,1]. Esta interpolación debe hacerse, naturalmente, de manera que los valores intermedios no salgan de este rango.

Comenzamos eligiendo como primeras coordenadas de $\{P_i\}$ los valores enteros 1,2,... ,m; de esta manera podemos representar tal función así:



Esto nos lleva a definir conceptos análogos a los de teoría ondulatoria tradicional, como *amplitud* (que vendrá dado por el rango de la curva funcional obtenida) o *longitud de onda*, que será la interdistancia entre puntos consecutivos, considerando esta como una cantidad constante. En este caso inicial, esta longitud es 1. La *frecuencia* será, directamente, la inversa de la longitud de onda.

En cuanto a la elección de los valores en $\{P_i\}$, los hemos recorrido elegido en un intervalo [0,1] de manera equiprobable, pero podríamos haber escogido cualquier función de probabilidad. Es más, un método habitual de escoger estos números es el utilizado para generar plasmas y superficies de Mandelbrot (su versión tridimensional):

1. Se determinan valores aleatorios en los extremos del intervalo, o bien en los vértices de una rejilla. El objetivo es calcular las intensidades en los demás puntos del espacio. Elegimos un número d de subdivisiones (habitualmente d=2), y una cantidad H que determina la variación en cada iteración. Habitualmente, en torno a $1/d$. Hacemos i=1 para comenzar el proceso iterativo.

2. Se subdivide el intervalo/rejilla en *d* partes, obteniendo nuevos puntos a los que les asignamos los valores por interpolación de algún tipo, respecto de los vértices originales adyacentes.
3. Recurrimos a la función de ruido (puede ser una cantidad aleatoria), o cualquiera de las que se explican más abajo. El valor obtenido se multiplica por $(1/d)^{iHd}$
4. Volvemos a 2 si los pixeles del dibujo no se han completado.

El proceso que pretendemos explicar es mecánicamente parecido, pero permite generalizar la generación de puntos iniciales y el uso de las funciones de ruido.

Funciones de ruido Perlin

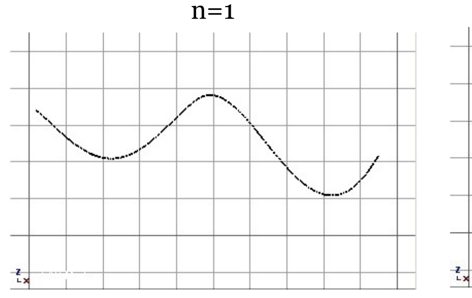
Ken Perlin creó las funciones que llevan su nombre sumando funciones de interpolación de ruido sucesivas, f1,f2,..., fn, cuyas amplitudes disminuyen progresivamente, mientras que las frecuencias van aumentando. Estas variaciones son geométricas, es decir, frecuencia y amplitud cambiarán en cada nueva función según factores multiplicadores constantes.

Llamemos **lacunarity** y **fracDim** (*lagunaridad* y *dimensión fractal*) a las variables que representan, respectivamente, a estas cantidades. A estas funciones las llamaremos *octavas*, y darán el grado de detalle deseado a la textura. La función resultante queda, entonces:

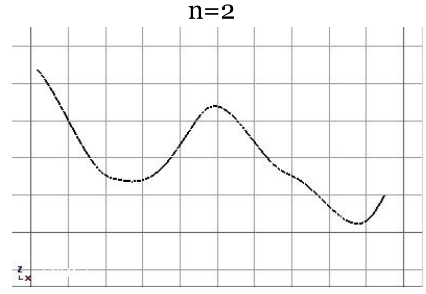
$$P_n(x) = \sum_{i=1}^{n} \{ fracDim^i \cdot f_i(x \cdot lacunarity^i) \}$$

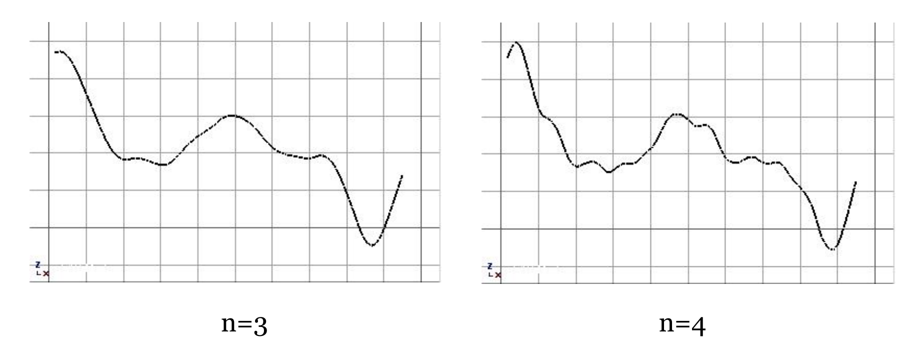
Por ejemplo, tomando **lacunarity**=2 y **fracDim**=0'5, podemos obtener un sucesión de funciones Perlin (según el número n de octavas) como esta:

n=1



n=2





n=3 n=4

Finalmente, podemos comparar dos funciones en la quinta octava variando los parámetros:

fracDim=0.5
lacunarity=2

fracDim=0.7
lacunarity=2

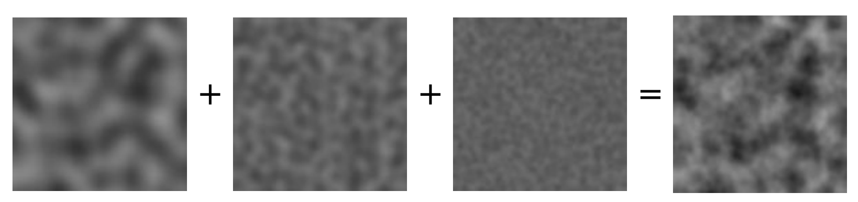
fracDim=0.5
lacunarity=2.1

fracDim=0.7
lacunarity=2.1

El dr. Forest Kenton (Ken) Musgrave, cuyo método se comenta más abajo, usó el término *persistence* para referirse directamente a la amplitud de cada sumando de la función de Perlin, de manera que:

amplitud = **persistencia**^i

Sin ninguna dificultad, podemos extender esta forma de trabajar a dominios en $\mathbb{R}^n$, pudiendo obtener con este proceso, por ejemplo, un valor numérico para cada punto en el espacio bi- o tridimensional real. Por ejemplo, dado un punto en una superficie que queremos texturizar, podemos aplicar un sencillo algoritmo tipo Perlin para obtener hasta la tercera octava, obteniendo como suma esta textura:



Ruidos en Blender 3D

Perlin, tal cual ha sido definido, no es la única manera de crear ruidos en Blender, aunque sí la más adecuada para crear paisajes y eventos de carácter natural: terrenos, texturas nubosas, superficies líquidas... Existen métodos para dibujar escenas pseudoaleatorias que nos permiten crear entes más o menos regulares, basados en la misma suma de funciones de interpolación sobre puntos al azar, solo que ya no son tales funciones, sino algoritmos que generan otros tipos de funciones, y que pueden requerir más parámetros. Discutiremos la definición y uso de la triangulación de Voronoi, sus parámetros (constantes de Worley), y los resultados de aplicarlo sobre distintas definiciones de distancia:

- euclídea $d=\sqrt{x^2+y^2+z^2}$
- euclídea al cuadrado $d=x^2+y^2+z^2$
- Manhattan $d=|(x)|+|(y)|+|(z)|$
- n-Minkovsky $d=\sqrt[1/n]{|x|^n+|y|^n+|z|^n}$

  (generaliza las anteriores)

y se verá también el método de *Musgrave*, usado por **Terragen** y el generador de mundos del propio autor, **MojoWorld**, para crear paisajes y nubes.

Terrynoise

Cuando Blender no disponía de texturas generadas por ruido, **Alfredo De Graaf**, conocido como **eeshlo** en nuestros foros, escribió una librería en C que contenía funciones para generar números aleatorios y basados en algoritmos como los explicados anteriormente. El objeto de esta librería, llamada *dynoise*, era usarla como base para el proyecto Dynamica, con el que se pretendía crear un motor de partículas y simulador de tejidos físicamente creíbles. Dynamica fue abandonado, pero dynoise seguía disponible para quien quisiera acceder a ella desde sus scripts.

Fue en esa época cuando programé *Terrynoise*, para aprovechar esta capacidad que tan lejos quedaba para Blender por aquel entonces.

De las funciones disponibles, Terrynoise utilizaba:

- MultiFractal
- Rigged Multifractal
- Hybrid Multifractal
- HeteroTerrain
- cellNoise
- Turbulence
- vlNoise
- Noise
- Random

El script actuaba sobre los vértices aplicando en la coordcenada **z** una cantidad generada por la función elegida por el usuario. La primera funcionalidad consistió en seleccionar qué vértices iban a ser afectados por el cálculo y en qué medida. Esto se consiguió usando el grupo de vértices llamado 'DYNOISE', en homenaje a la librería de eeshlo.

Blender integró en la versión 2.33 las funciones en su código, además de los nuevos tipos de textura procedurales:

- **Distorted Noise** (calcula ruido usando como lagunaridad el valor de otro)
- **Voronoi** (algoritmo de Voronoi de cercanía, con sus variantes W1-4)
- **Musgrave** (aplicar la síntesis espectral al ruido, previamente ampliado e interpolado)

lo que, unido a los ya existentes (Clouds, Marble, Wood) creó una nueva

forma de generar texturas con la nueva gama de ruidos:

- Cellnoise
- Voronoi Crackle
- Voronoi F2-F1
- Voronoi F4
- Voronoi F3
- Voronoi F2
- Voronoi F1
- Improved Perlin
- Original Perlin
- Blender Original

lo que supuso un gran avance en la creación de materiales en Blender. Esto, unido a la implementación de mapas de desplazamiento en la versión 2.32, hizo que *Terrynoise* fuese de repente obsoleto, ya que fue incluso superado por la existencia de la opción 'simple subdivision', que permite usar el mapa de desplazamiento sobre los vértices generados por subdivisión en tiempo de render.

La ventaja, sin embargo, reside siempre en la visualización del ruido durante el modelado, ya que se crea una nueva malla modificada, lo que permite exportarla tal cual, o incluso usarla en el motor de juegos de Blender.

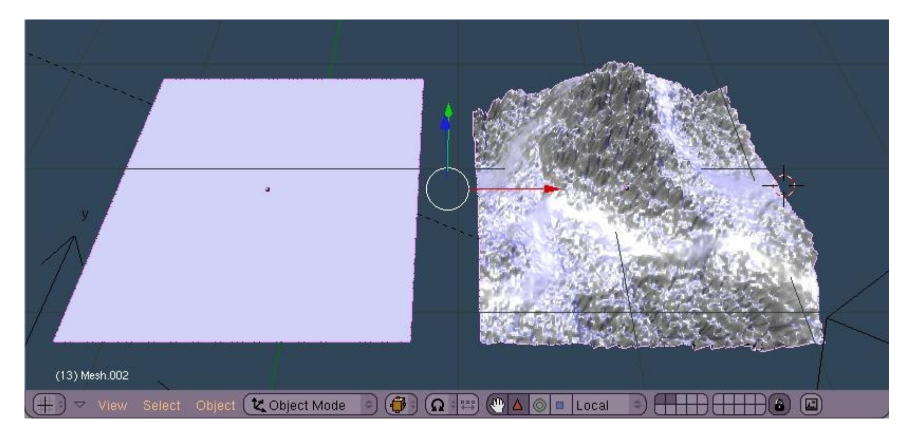
La incorporación de texturas procedurales hizo que pronto los desarrolladores del API de Python programaran un acceso a estas caracteríscas funciones de ruido mediante métodos en una nueva librería, *Blender.Noise*.

El paso siguiente fue obvio: programar la posibilidad de que los vértices fueran afectados a lo largo de sus normales, para producir efectos de ruido sobre objetos ya existentes, no sólo levantar meros paisajes sobre planos.

La reciente intervención de **Mathias Panzenböck** (**panzi**) en el código fuente fue decisiva: realizó una limpieza de código, e implementó la posibilidad de que el usuario entrara sus propias funciones matemáticas implícitas, usando como variables las coordenadas de los vértices y los operadores estándar, junto a las funciones matemáticas incluidas en la librería *math* de Python.

Panzi también se encargó del tratamiento de ficheros para grabar y salvar parámetros.

Gracias a estas mejoras *Terrynoise* recibió un nuevo impulso, completando sus capacidades más allá de sus aspiraciones iniciales, implementando



entre otras nuevas mejoras la posibilidad de animar la proyección del ruido sobre la malla y su intensidad, consiguiendo efectos de burbujeo, oleaje, vuelos sobre terreno, o zoom, además de introducir el tiempo como nueva variable para las funciones matemáticas.

Finalmente, es de reseñar que puede aplicarse sobre cualquier número de mallas, existiendo un objeto activo sobre el que se trabaja, lo que permite además iterar ruidos si lo aplicamos sobre una malla previamente modificada. Esto se consigue usando las propiedades que existen en el motor de juegos para cada objeto, usándolas como variables de configuración.

Psicofisiología de la percepción

Fernando Arroba Rubio
gnotxor@gmail.com ::



Cuando hablamos de percepción solemos tender a pensar inmediatamente en la percepción visual. Como sistema perceptivo dominante es el que analizaremos a continuación, tanto desde un punto de vista funcional como psicológico.

El ojo es una esfera que cumple con la estructura de una cámara oscura. Sus funciones son tanto ópticas como sensoriales. En su funcionalidad óptica se comportaría como una cámara fotográfica: su cometido es recibir la luz, a través del iris (diafragma) y enfocarla con su sistema de lentes compuesta de córnea y cristalino, ajustando los rayos luminosos sobre la retina (película sensible).

El sistema visual

La esclera o esclerótida correspondería con la caja o cuerpo de la cámara. (Hay que ver lo original que soy haciendo estas analogías). Aunque en realidad la anología mejor no es la cámara fotográfica sino la de video.

Atravesamos las estructuras en la misma dirección que lo haría la luz y veremos su funcionalidad de manera rápida.

El ojo está estructurado en tres capas: la superficial o fibrosa, que comprende la esclera y la córnea; la túnica vascular, tracto uveal o úvea, que comprende la coroidea, el cuerpo ciliar y el iris; y la túnica interior o retina, que incluye la porción fotosensible de la pared posterior del ojo.

La primera estructura que nos encontramos es la córnea. Tras atravesarla nos encontramos en la cámara ocular, un espacio lleno de líquido entre la córnea y el cuerpo vítreo. Este espacio se encuentra dividido en dos (cámara anterior y cámara posterior) por el iris. El humor acuoso que lo llena es el resultado de un ultrarrefinado de la sangre y su función es aportar nutrientes y oxígeno a la lente y a la córnea, que son avasculares.

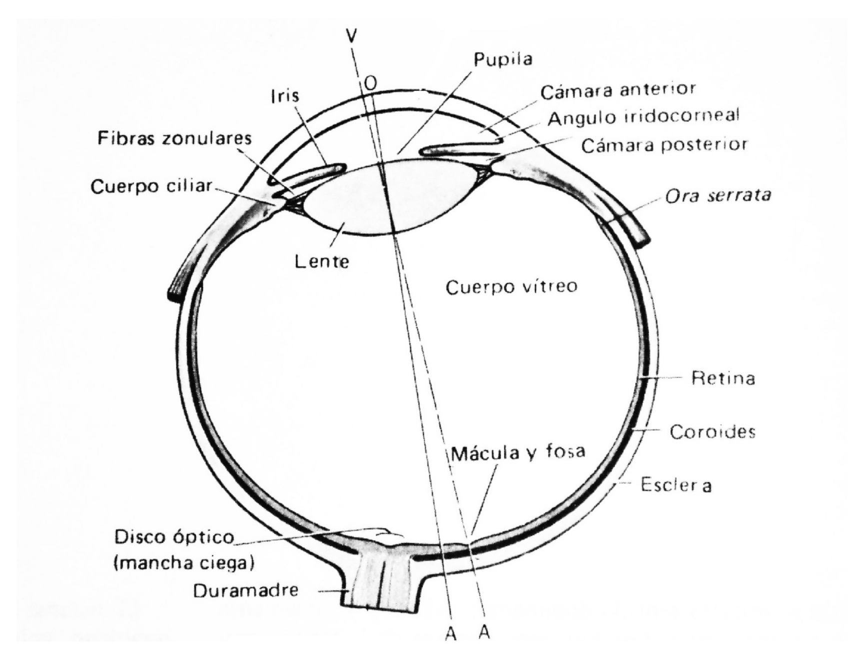


Figura 1. *Corte transversal del ojo, vista sagital.*

Por último, en la parte posterior nos encontramos la retina. Es el elemento fotosensible. Podemos distinguir en la retina varias formas o estructuras especiales. Destacando el "dico óptico" y la "mácula". El primero es el punto ciego de la retina y la segunda presenta una fosa que es la zona más sensible a la luz.

Aspectos funcionales de la retina

Podemos considerar la retina como la parte móvil del cerebro. Es una estructura del sistema nervioso central que se mueve con el ojo. No entraremos en las subestructuras que la componen sino en la funcionalidad de la misma, comenzando con los fotorreceptores, los "conos" y los "bastones".

Bastones y conos

Tanto los conos como los bastones son células receptoras alargadas. Ambos tipos de células difieren anatómicamente según se encuentren en el centro de la retina o en la periferia. Los bastones pueden medir 2 μm de diámetro en la zona central de la retina y llegar en las periféricas a 4 μm ó 5 μm. Los conos de la fosa miden 1,5 μm mientras que los de las zonas que la rodean alcanzan de 5 a 8 μm.

La rodopsina es el fotopigmento de los bastones. Los conos, por su parte, pueden contener tres fotopigmentos diferentes, sensibles a los 435 *nm* (azul) cianolabe, 535 *nm* (verde) clorolabe y 565 *nm* (rojo) eritrolabe.

La cantidad de energía radiante y las longitudes de onda son los factores físicos de la luz esenciales para el proceso visual. La cantidad de energía radiante se interpreta como "brillantez" mientras que las longitudes de onda son la base para la discriminación del color.

Los bastones son los sensores ligados a la percepción del blanco, gris y negro, respondiendo a todas las radiaciones del espectro visual e incluso a las de la banda ultravioleta. A diferencia de los bastones, los conos sólo son sensibles a diferentes longitudes de onda de manera selectiva.

La visión en color depende de dos procesos. El primero es la actividad receptora de tres tipos de conos. El segundo el procesamiento dentro del sistema nervioso central. Del primer proceso se sabe mucho y está bien estudiado, sin embargo del segundo se desconoce prácticamente todo.

Teoría dual de la visión

La teoría dual de la visión se basa en la concepción de la retina como si fuera un mosaico de "cuatro" tipo de receptores (los bastones y los tres tipos de conos). Los bastones son sensibles a la luz de baja intensidad y los conos se especializan en la visión en color y en el registro de detalles finos.

Las modernas variante de esta teoría no hacen incapié en la distinta funcionalidad de conos y bastones, sino más bien en la interacción entre conos, bastones y otras neuronas retinianas.

Sin embargo, el nombre de esta teoría evoca otra dualidad curiosa. La dualidad de la luz, como "partícula" y como "onda". Parecería que los bastones son sensibles a los fotones como partículas mientras que los conos lo serían como ondas.

Campos receptores de la retina

En la retina de cada ojo existen unos 120 millones de bastones, 7 millones de conos y algo más de un millón de células gaglionares cuyas fibras forman el nervio óptico. La desigualdad entre el número de receptores y de células "transportadoras" evidencia el fenómeno de convergencia que opera en la retina. Esta convergencia es lo que conocemos como campos receptores. Un campo receptor es la superficie que cubren los receptores que estimulan al mismo neurocito ganglionar.

La *fosa* o *mácula* carece de bastones y contiene entorno a los 4.000 conos y la misma cantidad de neurocitos. En ella, el campo receptor se corresponde con los 2 μm aproximadamente, lo que corresponde con un ángulo corneal de sólo unos pocos minutos. Esta zona es la región de mayor discriminación de la retina y la parte del objeto percibido con más detalle es el que se proyecta sobre esta zona.

En la retina periférica un campo receptor puede tener hasta $1mm$ de diámetro, lo que corresponde con un arco de 3^{o} en el campo visual (que abarca 160^{o}).

Estos campos receptores extrafoveales con sensores mezclados (conos y bastones) no son muy precisos, pero a cambio pueden trabajar con estimulaciones y umbrales más bajos.

Aspectos funcionales de la visión

Para nosotros es imposible mantener los ojos quietos un solo instante, ni aún cuando concentramos nuestra mirada en algo, siempre se producen pequeños movimientos. Esta especie de "barrido" de alta frecuencia es esencial para mantener una imagen visual. Cuando experimentalmente se consigue mantener la proyección de una imagen sobre la retina de forma estática, la misma tiende a desvanecerse, a disminuir su coloración y a que sus contornos se desvanezcan, como se puede advertir en la figura 2.

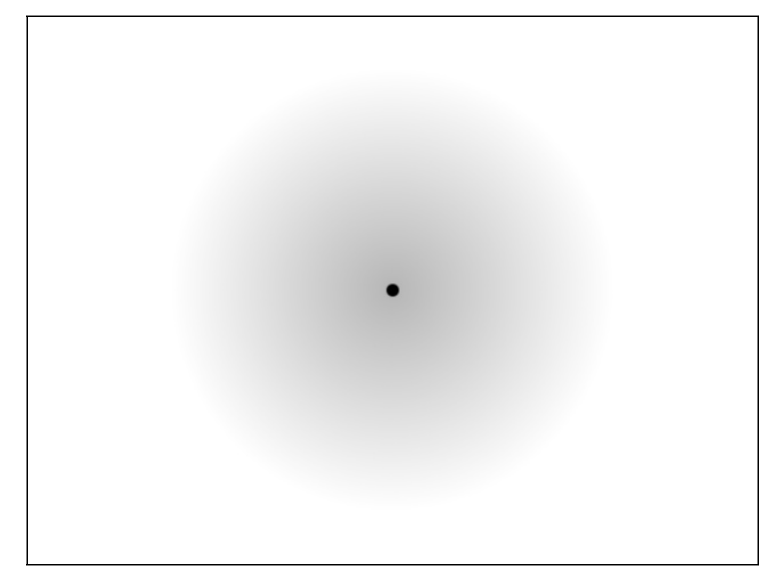


Figura 2. *Si fijamos la vista en el punto central la "corola" parece difuminarse.*

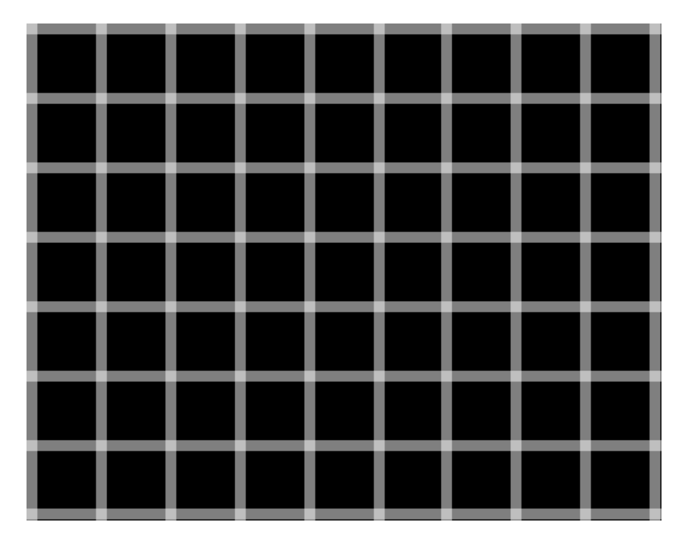


Figura 3. *El efecto de inhibir los bordes tiene como resultado la aparición de puntos negros* fantasmas.

Agudeza de contornos y contraste

En la percepción de los contornos se encuentra implicado un fenómeno perceptivo que se conoce como *inhibición lateral*. El fenómeno es un mecanismo de interacción entre las neuronas para enfatizar los límites. Por ejemplo, si colocamos un folio blanco sobre una zona negra. Los efectos de esta peculiaridad se pueden apreciar en la figura 3.

La percepción de los contornos, del color y del contraste, es por tanto, una apreciación distorsionada desde la misma recepción del estímulo. Algunos efectos son llamativos como el del ejemplo de Edward H. Adelson, que se puede apreciar en la figura 4. En dicha figura los cuadros *A* y *B* son exactamente el mismo tono de gris, sin embargo, el entorno y los contrastes con las zonas adyacentes hacen que se interpreten como cromatismos diferentes.

Adaptación, agudeza visual y visión cromática

El ojo no es sensible siempre a la misma cantidad de luz. Un ojo adaptado a la oscuridad puede ser estimulado con una diezmilésima parte de la energía que estimularía al mismo ojo adaptado a la luz. Sin embargo, la velocidad a la que se adapta el ojo es variable. Se considera que se tarda una media hora en que el ojo se adapte a la oscuridad, mientras que la adaptación a la luz se realiza en pocos segundos.

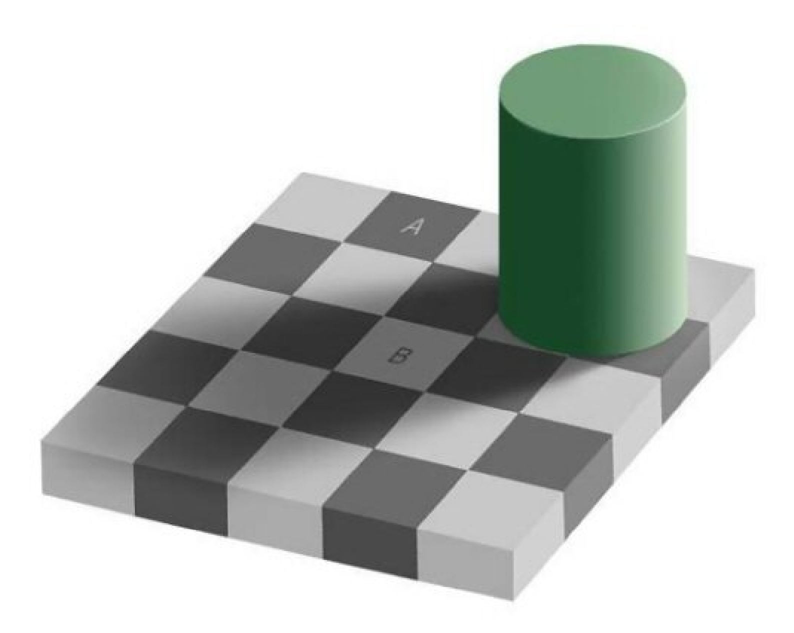


Figura 4. *Los cuadros marcados con la A y la B presentan el mismo matiz de gris sin embargo, son percibidos de manera distinta por el ojo.*

La capacidad que tiene el ojo para adaptarse a la oscuridad no depende sólo de la cantidad de luz. Se ha comprobado que determinados estados de ánimo, la cantidad de oxígeno en sangre y otros factores influyen en la adaptación de la pupila.

La agudeza visual del ojo es la medida en que puede distinguir detalles. Dicho de otro modo: la agudeza visual la podemos medir como la distancia mínima entre dos puntos para que sean percibidos como tales y no como una sola forma. Por supuesto la zona de mayor agudeza de la retina es la fóbea.

Clínicamente la agudeza visual se mide con una fracción cuyo numerador es siempre 20. Ese numero representa el número de pies (6 metros aproximadamente) que se toma por referencia. Es decir, una persona con una agudeza visual normal tendría un valor *20/20*, es decir a veinte pies puede distinguir letras que se distinguen a 20 pies. Si tiene una agudeza visual de *20/100* quiere decir que está distinguiendo a 100 pies las letras que una persona normal distinguiría a 20. Si la agudeza visual es de *20/15* quiere decir que distingue las letras a tan sólo 15 pies.

La visión cromática es la interpretación que hace el cerebro de la recepción de distintas longitudes de onda y es por tanto una experiencia subjetiva. La interpretación de la luz amarilla cuando se estimulan simultáneamente

receptores rojos y verdes se considera que es consecuencia del procesamiento cortical. La percepción del color es algo limitado a algunos vertebrados diurnos. Parece que durante la filogenia la percepción del color evolucionó de forma independiente muchas veces.

El ***tono*** de un color es la medida de la longitud de onda que se percibe; el rojo, verde, azuly amarillo son tonos diferentes.

El ***brillo*** de un color es la sensación subjetiva determinada por la cantidad de negro en el color. El más negro es el menos brillante, es decir, a mayor cantidad de negro se absorbe una mayor cantidad de luz y por tanto se refleja una cantidad menor.

La ***saturación*** o *pureza de un color* es la sensación subjetiva determinada por la cantidad de blanco dentro del mismo. El blanco altera el tono debido a que está compuesto por colores. Cuanto menos blanco tenga se dice que el color está "más saturado" o es "más puro su matiz".

El efecto Purkinje

El efecto Purkinje debe su nombre a dicho experimentador que observó que había diferencias de percepción cromática en situaciones de poca intensidad lumínica. Observó que las flores ligeramente azules lo eran más a la caída de la tarde, mientras que en el crepúsculo las flores rojas parecían prácticamente negras. De hecho, esa es la explicación para la abundancia del color rojo en el fondo marino, muchos peces, corales y animales presentan dicho color para camuflarse en la oscuridad.

El ojo adaptado a la oscuridad es más sensible al verde, mientras que el adaptado a la luz lo es más al amarillo.

Postimágenes

Las postimágenes son sensaciones ópticas que persisten después de presentado el estímulo y pueden ser de dos tipos: las **postimágenes negativas** y las **postimágenes positivas**.

Las *postimágenes negativas* se producen cuando miramos un estímulo durante unos 10 segundos y después fijamos la mirada en una superficie blanca. La imagen que se produce es *complementaria* al estímulo original. Tenemos un ejemplo en la figura 5.

La *postimagen positiva* se produce cuando un estímulo iluminado intensamente se presenta durante 2 ó 3 segundos y después se cierran los ojos. Esta postimagen se parece al estímulo original en color y forma, aunque no en intensidad.

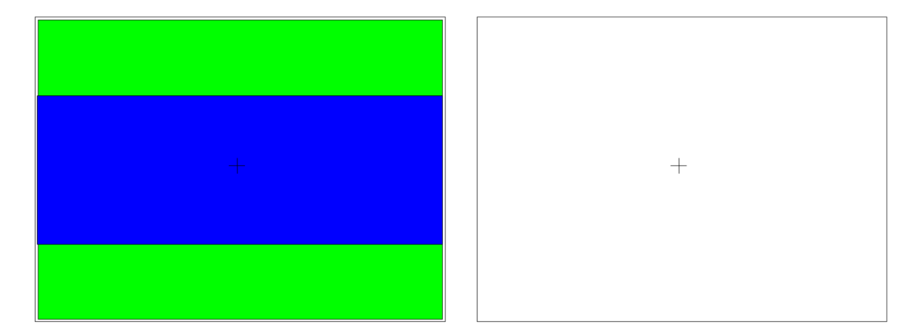


Figura 5. *Mirar fijamente a la cruz durante unos 10 segundos y después fijar la mirada en la cruz de la derecha, la postimagen formada debería corresponder con los colores de la bandera española*[1].

Aspectos psicológicos de la percepción

Uno de los factores que más influyen en la percepción es **la atención**. Todos conocemos la diferencia de actitud que hay entre ver y mirar o entre escuchar y oír. La *atención* es pues uno de los primeros factores psicológicos que van a modificar la percepción.

La atención dista bastante de ser una forma de información pasiva. Al contrario, consiste en modificar el estado de vigilancia sobre determinados estímulos, de forma que aumenta el nivel de "alerta" que producen los mismos. La cantidad de estímulos a los que puede atender el cerebro humano de forma simultánea está limitada a un rango de 6 a 11, según los casos. Esta limitación se ve compensada por la concurrencia de otras características o propiedades funcionales.

Esta limitación de amplitud exige que se haga una selección de estímulos juzgando la relevancia de unos y otros. En esencia, sirven para establecer esta relevancia factores como el contraste, el tamaño, la intensidad y el movimiento de los objetos o estímulos.

Leyes de la percepción

Las llamadas "leyes de la percepción" fueron formuladas en su mayoría por los llamados "psicólogos de la forma" o *Gestalt*. El gran número de éstas podemos reducirlo a tres grupos fundamentales: las que se refieren a la separación entre figura y fondo, las que aluden a las propiedades de las

1 Nota del maquetador: Suponiendo que la impresión de este documento que tienes en tus manos es en color. :-) En la imagen a color, las bandas exteriores son de color verde y la central de color azul.

totalidades figurales y las que precisan las condiciones por las que se agrupan estímulos en figuras.

Por ejemplo, *Helson* definió cinco principios:

1. **Ley de la primacía**, en virtud de la cual los todos son primarios y aparecen con prioridad a las partes.
2. Percibir todos es más natural que percibir partes. **Ley de la pregnancia**.
3. Los todos (figuras) tienden a articularse de la forma más completa, simétrica, sencilla y perfecta posible. **Ley de la buena figura**.
4. La **ley de la autonomía** que los todos tienden a ser regulados por factores intrínsecos, más que por factores externos a ellos.
5. Finalmente, las partes derivan sus propiedades de su posición, o función en el todo. De ahí la posibilidad de que un mismo estímulo sirva de base a la percepción de figuras diferentes, como en el caso de las figuras reversibles. **Ley de la flexibilidad del contorno**.

Las leyes que rigen la **agrupación de los estímulos** mencionan los siguientes factores:

1. **Proximidad**: a igualdad de circunstancias, los estímulos más próximos tienden a percibirse como formando parte de un mismo objeto.
2. **Semejanza**: a igualdad de circunstancias, los estímulos más semejantes tienden a percibirse como formando parte de un mismo objeto.
3. **Continuidad**: a igualdad de circunstancias, tendemos a percibir como formando parte de una misma figura los estímulos que guardan entre sí una continuidad de forma.
4. **Simetría**: la tendencia a organizar los estímulos en una forma simétrica, puede competir con alguna de las anteriores leyes, por ejemplo, con la de la semejanza, y configurar unitariamente estímulos heterogéneos.
5. **Constancia**: la tendencia a precibir los objetos exteriores de una manera estable. Por ejemplo, si observamos un objeto a un metro de distancia y nos separamos de él, su imagen en nuestra retina se hace más pequeña, sin embargo seguimos percibiendo sus demensiones de manera constate.

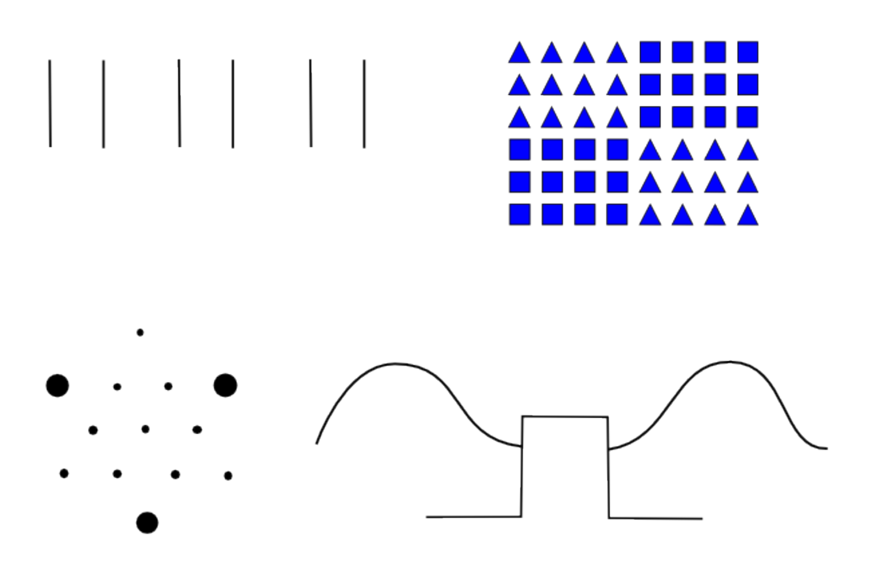


Figura 6. *Algunas figuras que ejemplifican las leyes de la percepción.*

El número de las leyes de la percepción excede con mucho las expuestas aquí. Sería imposible en este espacio de tiempo hablar de todas ellas. Éstas, las expuestas por la Gestalt, junto con algunas otras como la de la *constancia perceptiva*, constituyen el cuerpo básico de las leyes de la percepción siendo las demás meros derivados o incluso redundancias de las mismas.

La Matemática de los Gráficos 3D

Juan David González Cobas
coba@epsig.uniovi.es ::



Este documento aporta un repaso de matemáticas pertinentes a las tareas que afrontan cada día artistas y desarrolladores 3D. Se incluyen aspectos de la geometría de los gráficos3D, la representación de objetos y sus transformaciones geométricas.

Sistemas de coordenadas

La descripción de escenas tridimensionales requiere usualmente el empleo de un sistema de coordenadas cartesiano. Los puntos del espacio quedan determinados de forma única por sus tres *coordenadas cartesianas (x, y, z)*.

Hay otros sistemas que no se emplean tanto en gráficos 3D. Los más corrientes son las coordenadas *cilíndricas y esféricas*.

Los tres sistemas asignan coordenadas a los puntos de acuerdo al esquema de la figura 1. Cada sistema usa tres de las dimensiones que allí se referencian:

cartesianas: (x,y,z)
cilíndricas: (ρ,ϕ,z)
esféricas: (r,θ,ϕ)

Las ecuaciones que nos permiten cambiar entre un sistema y otro son:

Cartesianas ↔ Esféricas

$$x = r \sin \theta \cos \phi$$

$$y = r \sin\theta \sin \phi$$

$$z = r \cos \theta$$

Cartesianas ↔ Cilíndricas

$$x = \rho, \cos \phi$$

$$y = \rho \sin \phi$$

$$z = z$$

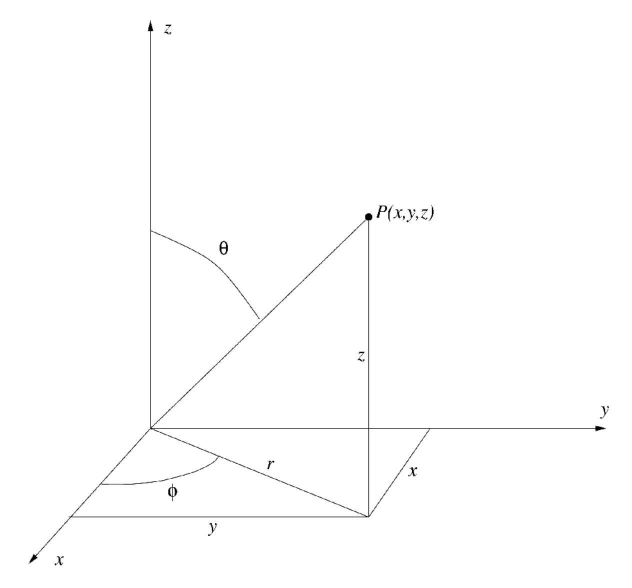


Figura 1. *Sistemas de coordenadas cartesianas, cilíndricas y esféricas.*

De mayor importancia en gráficos por computador son las coordenadas homogéneas o proyectivas. Los puntos ordinarios del espacio tridimensional reciben cuatro coordenadas en lugar de tres:

$$(x,y,z) \leftrightarrow (x,y,z,w)$$

Esto introduce una redundancia evidente, de forma que a cada punto del espacio le asignamos infinitas cuaternas de coordenadas homogéneas, de acuerdo con la equivalencia:

$$(x, y, z, w) \equiv (x', y', z', w') \Leftrightarrow$$
$$\alpha\,(x, y, z, w) = (x', y', z', w') \text{ para algún } \alpha \neq 0$$

de forma que cuaternas proporcionales denoten el mismo punto. La terna usual *(x, y, z)* se identifica con la cuaterna *(x, y, z, 1)*, y la cuaterna *(x, y, z, w)* denota un punto *(x/w, y/w, z/w)* del espacio ordinario.

Si w=0, tenemos un punto del infinito; lo que logramos con la introducción de cuaternas es, precisamente, una coordinatización de la geometría del espacio proyectivo tridimensional. Esta es la primera ventaja de la representación: no se necesita ningún tratamiento especial para puntos del infinito casos especiales de paralelismo. Las transformaciones proyectivas se traducen fácilmente a transformaciones lineales en el espacio de 4-tuplas (es decir, a matrices 4 *x* 4). Otra ventaja es que las transformaciones afines usuales también se tratan de forma regular como productos de matrices, como veremos en la sección 4.

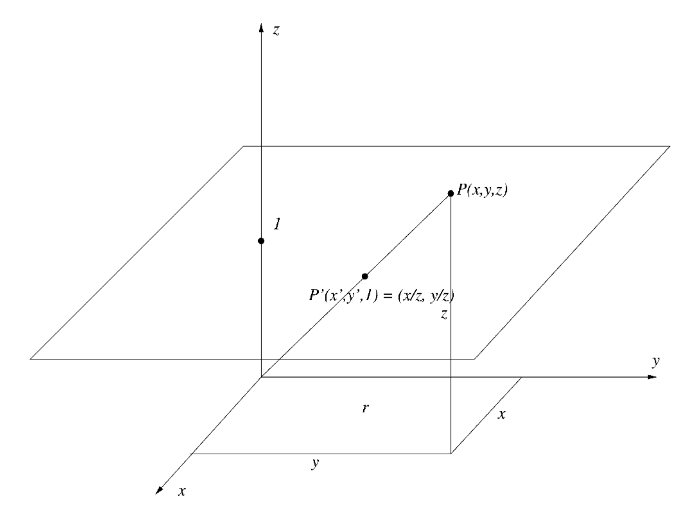


Figura 2. *Coordenadas Proyectivas.*

Álgebra vectorial

Operaciones con vectores

Los vectores del espacio tridimensional se representan por lo común por sus coordenadas cartesianas, y las operaciones con vectores se definen en términos de ellas:

adición

(x, y, z) + (x', y', z') = (x + x', y + y', z + z')

substracción

(x, y, z) - (x', y', z') = (x - x', y - y', z - z')

escalado

λ (x, y, z) = (λ x, λ y, λ z)

producto escalar

(x, y, z) × (x', y', z') = xx' + yy' + zz'

norma

$$\|(x, y, z)\| = \sqrt{v \cdot v}\, \sqrt{[x^2 + y^2 + z^2]}$$

producto vectorial

(x, y, z) × (x', y', z') = (x, y, z) ∧ (x', y', z') = (yz' - zy', zx' - xz', xy' – yx')

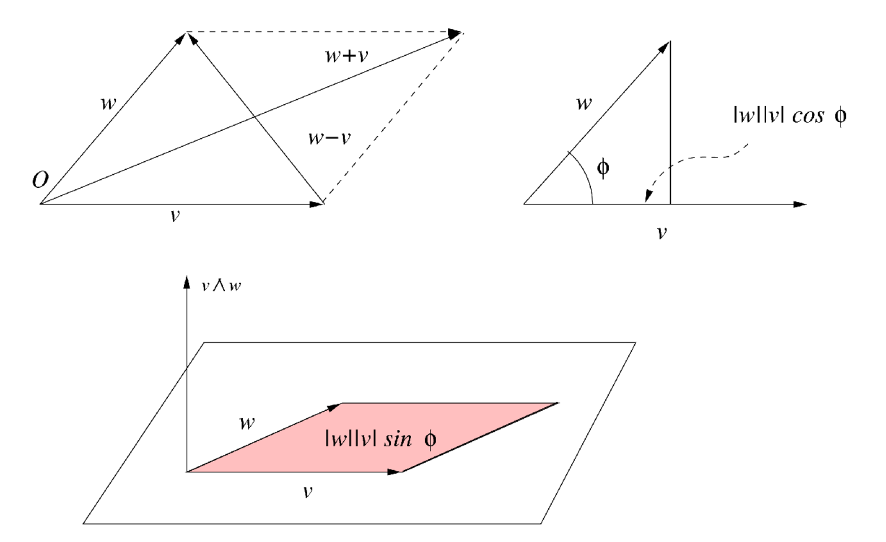


Figura 3. *Interpretación geométrica de las operaciones con vectores.*

Interpretaciones geométricas

La interpretación geométrica de estas operaciones puede verse en la figura 3. En ella vemos que el producto escalar tiene la interpretación alternativa

$$v\cdot w=\|v\|\|w\|\cos\phi$$

y el producto vectorial es ortogonal a los factores, con módulo igual al área del paralelogramo que definen:

$$v\times w=\|v\|\|w\|\sin\phi$$

De esto deducimos que los dos productos son de especial interés, al permitirnos calcular los elementos métricos de un modelo: distancias, ángulos, área, volúmenes, ortogonalidad y paralelismo. Veamos algunos ejemplos:

- La norma de un vector $v = (x, y, z)$ (su longitud) se calcula por medio del producto escalar: $\|v\|=\sqrt{v\cdot v}$
- La distancia entre dos puntos es la norma del vector que los une:
  P = (x, y, z)
  Q = (x', y', z')
  P Q = (x' - x, y' - y, z' – z)
  $$d(P,Q)=\|P,Q\|=\sqrt{(x'-x)^2+(y'-y)^2+(z'-z)^2}$$

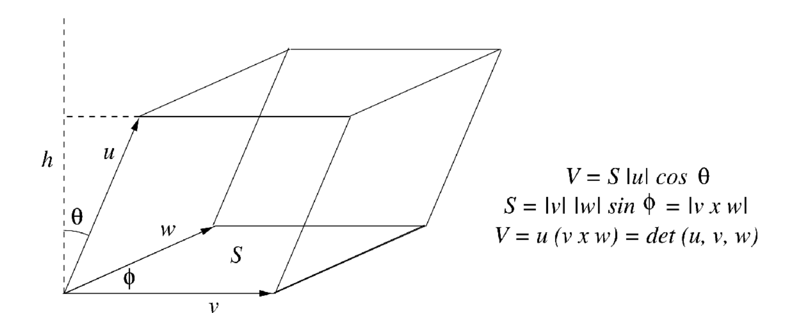


Figura 4. *Volumen de un paralelepípedo.*

- El ángulo f entre v y w es $\cos\Phi = \frac{v \cdot w}{(\|v\| \cdot \|w\|)}$
- Dos vectores son ortogonales si y sí y sólo si $v \times w = 0$
- Un vector normal al plano definido por *v* y *w* viene dado por $v \times w$
- El volumen del paralelepípedo (celda) definido por los vectores *u*, *v*, *w* (ver figura 4) es

$$det(u, v, w) = u \cdot (v \times w) = \begin{pmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{pmatrix}$$

Esta cantidad se conoce también como el triple producto de los vectores *u, v, w* o, más corrientemente, su *determinante*. El cómputo del mismo sigue reglas bien conocidas en las que no nos detendremos demasiado.

Bases

Dado cualquier conjunto de vectores $\{v_1, v_2, ..., v_n\}$, el subespacio engendrado por ellos es el conjunto de todos los vectores que podemos construir como combinación lineal de $\{v_1, v_2, ..., v_n\}$, usando la suma y la multiplicación por escalares:

$$\langle v_{1,} v_{2,} \ldots, v_n \rangle = \{ w | w = \lambda_1 v_1 + \lambda_2 v_2 + \ldots + \lambda_n v_n \}$$

El conjunto $\{v_1, v_2, ..., v_n\}$ es libre, o linealmente independiente, si

$$\lambda_1 v_1 + \lambda_2 v_2 + \ldots + \lambda_n v_n = 0 \Leftrightarrow \lambda_1 = \lambda_2 = \ldots = \lambda_n = 0$$

o, de otro modo, si ningún vector del conjunto puede expresarse como una combinación lineal de los demás. Un conjunto libre que genera el espacio total en consideración es una base del mismo. En nuestro espacio tridimensional, todas las bases tienen tres elementos. Además, una base B = $\{e_1, \ldots, e_n\}$ donde vectores diferentes son ortogonales

$$e_i \cdot e_j = 0 \; sii \; i \neq j$$

se denomina *ortogonal*. Si además

$$e_i \cdot e_j = 1$$

es decir, todos los vectores son de norma unidad, la base es *ortonormal*. En una base ortonormal, los vectores se pueden expresar de forma simple y conveniente como sigue:

$$x = (x \cdot e_1) e_1 + (x \cdot e_2) e_2 + (x \cdot e_3) e_3$$

Geometría afín y proyectiva

En la sección previa hemos mencionado de pasada la distancia entre puntos, y usado sin restricción la notación P,Q para referirnos al vector que une el punto P con el punto Q. Rigurosamente hablando, esto sólo es legítimo si al espacio tridimensional ordinario (carente de coordenadas) lo dotamos de una estructura afín.

Esto significa,simplemente, que existe una operación para trasladar cualquier punto P por cualquier vector v, de forma que el resultado sea otro punto que denotaremos por *P+v*. La operación de traslación debe satisfacer las propiedades obvias siguientes:

1. Para cada vector v, la correspondencia $P \rightarrow P + v$ es uno a uno.
2. Para cada punto *P*, se cumple $P + o = P$
3. Para cada punto *P* y vectores *v, w*, tenemos
$$(P + v) + w = P + (v + w)$$

Estas propiedades son tan obvias en el trabajo 3D ordinario, que las usamos sin conciencia de ello. La existencia de un vector que une P con Q, por ejemplo, es una consecuencia de ellas; la asignación de coordenadas en el espacio ordinario también lo es. Si añadimos a esto la existencia de un producto escalar, podemos definir conceptos de distancia, ángulos y ortogonalidad, de forma que cualquier problema de geometría métrica puede resolverse de forma analítica.

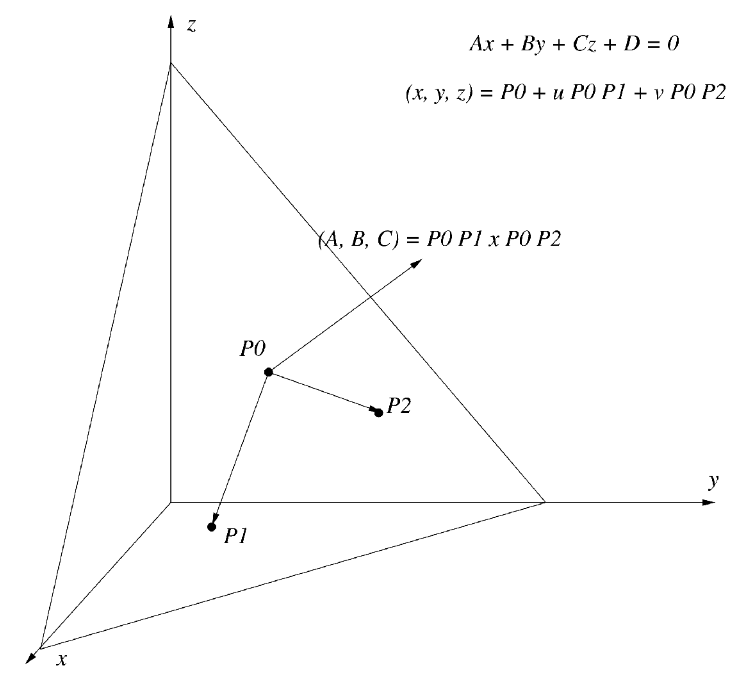


Figura 5. *Ecuaciones de un plano.*

Resumimos a continuación los hechos más simples relativos a entidades «rectas» en geometría afín tridimensional: puntos, líneas y planos. Estas son las variedad afines o lineales de la geometría habitual, y pueden definirse siempre por medio de sistemas de ecuaciones lineales.

Planos

Un plano se suele definir de dos maneras:

ecuación implícita

Por medio de una sola ecuación lineal que satisfacen las coordenadas de los puntos del plano

$$Ax + By + Cz + D = 0$$

ecuación paramétrica

Los puntos del plano se obtienen barriendo todos los posibles valores de dos parámetros reales *u* y *v*. En forma vectorial

$$P = P_0 + u , P_0 P_1 + v , P_0 P_2$$

o, en coordenadas

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + u, \begin{pmatrix} a \\ b \\ c \end{pmatrix} + v, \begin{pmatrix} a' \\ b' \\ c' \end{pmatrix}$$

Los vectores (a, b, c) y (a', b', c') son vectores de dirección del plano. Es fácil pasar de una representación a otras: a partir de la ecuación implícita es fácil obtener tres puntos no colineales P_0 , P_1, P_2 que nos dan las ecuaciones paramétricas. El paso de las ecuaciones paramétricas a la implícita requeriría, en principio, eliminar por reducción los parámetros *u,v*. Pero hay un método más fácil usando el producto vectorial:

$$(A, B, C) = P_0 P_1 \times P_0 P_2 = (a, b, c) \times (a', b', c')$$
$$D = -(A, B, C) \cdot P_0$$

Líneas

Las rectas pueden definirse por un punto y una dirección, o por la intersección de dos planos. Esto da lugar a las posibles representaciones analíticas de la recta con

ecuaciones implícitas.

Dos ecuaciones lineales (independientes) que definen la línea como intersección de dos planos:

$$A1x + B1y + C1z + D1 = 0$$
$$A2x + B2y + C2z + D2 = 0$$

ecuación paramétrica

Nos dan las coordenadas de puntos de la línea al barrer un paráetro real u todos los valores posibles:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = P_0 + u P_0 P_1 \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + u, \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

De nuevo, la conversión entre las dos representaciones es sencilla. La ecuación paramétrica puede ponerse en forma implícita eliminando el parámetro *u*; y de las ecuaciones implícitas obtenemos fácilmente dos puntos con los que construir la paramétrica; o un punto y el vector de dirección, proporcionado por la expresión

$$P_0P_1 = (A_1, B_1, C_1) \times (A_2, B_2, C_2).$$

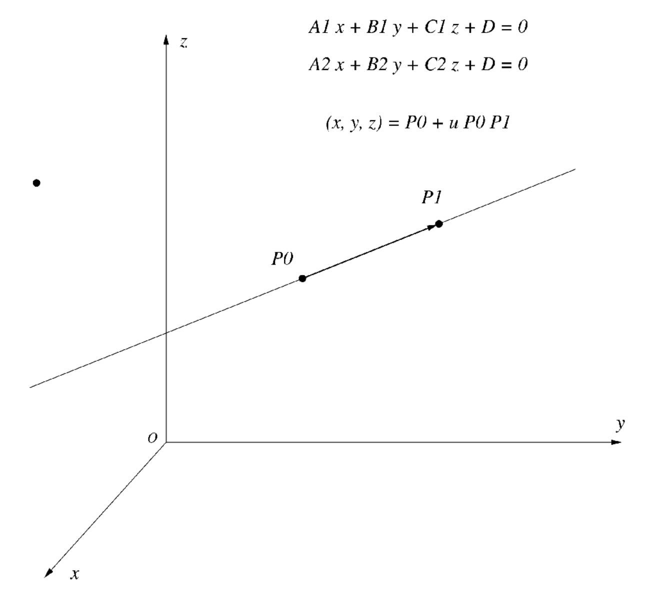


Figura 6. *Ecuaciones de una línea recta.*

Geometría de la incidencia y geometría métrica

Con estos conceptos, y teniendo en mente las propiedades del producto escalar y vectorial, podemos resolver cualquier problema de geometría analítica elemental. Mejor que una enumeración de problemas-tipo, vamos a resolver algunos problemas sencillos para adquirir una impresión de las técnicas que conllevan. Por ejemplo, queremos calcular la distancia entre dos rectas, como se muestra en la figura 7. Ambas rectas r_1 y r_2 vienen dadas por puntos P_i y vectores directores u_i. Un poco de reflexión nos convencerá de que la distancia mínima d entre las dos rectas se alcanza entre puntos Q_1 y Q_2 tales que el segmento Q_1Q_2 es ortogonal a las rectas dadas. Podemos ver esto más claramente construyendo los planos paralelos p_1 y p_2 que contienen a cada recta; entonces,Q_1Q_2 es perpendicular a ambos. Así pues

$$d=\left\| Q_1 Q_2 \right\| \quad \text{y} \quad Q_1 Q_2 \perp u_{1,} u_2$$

y entonces Q1Q2 = u1 × u2. Ahora, el vector P1P2 une puntos de ambas rectas, que son ortogonales a Q1Q2. Entonces, el producto escalar con el vector unitario u1 × u2 nos dará el módulo de la proyección de P1P2 sobre Q1Q2, que es exactamente d. Obtenemos

$$d=P_1 P_2 \cdot u_1 \times u_2 = det\left(P_1 P_{2,} u_{1,} u_2\right)$$

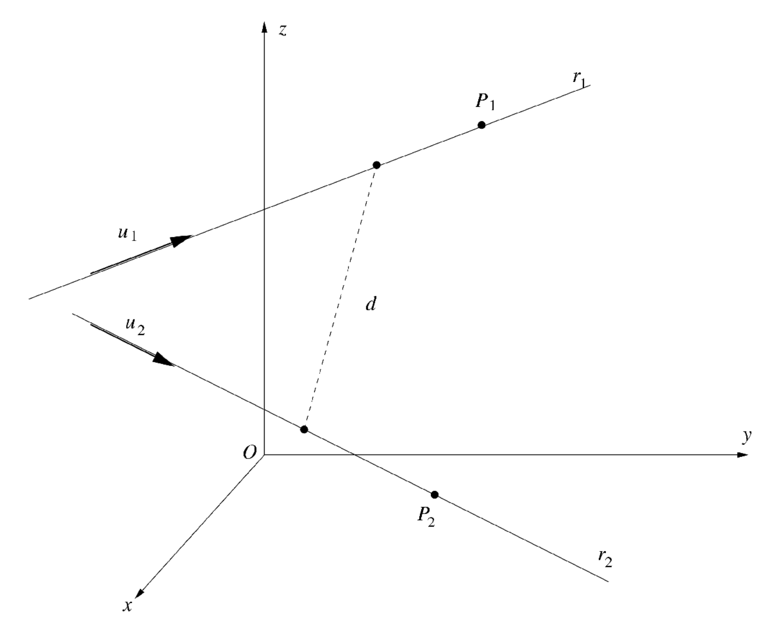


Figura 7. *Ejemplo de la distancia entre dos rectas.*

Matrices y transformaciones geométricas

Las tareas que podemos realizar con los conceptos vistos hasta ahora son bastante elementales. Un tratamiento más potente de los objetos geométricos exige poder expresar operaciones complejas con las formas. El método habitual de representar transformaciones lineales, y de calcular sus efectos, es el uso de matrices.

Una matriz $m \times n$ es una tabla rectangular de números de la forma

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \ddots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

que se escribe de forma más compacta como $(a_{ij})_{1\leq i\leq m, 1\leq j\leq n}$ o simplemente como (a_{ij}) si las dimensiones de la matriz se sobreentienden.

Al igual que los vectores, las matrices pueden sumarse y restarse

componente a componente. El producto de matrices viene definido por la regla

$$A=(a_{ij}) \qquad i=1\ldots m,\ j=1\ldots n$$

$$B=(b_{jk}) \qquad j=1\ldots n,\ k=1\ldots p$$

$$AB=(c_{ik}) \qquad i=1\ldots m,\ k=1\ldots p$$

y

$$C_{ik}=\sum_{j=1}^{n} a_{ij} b_{jk}$$

Observemos que, incluso cuando $m = n = p$, tenemos, es decir, el producto de matrices no es conmutativo. La transpuesta de una matriz se obtiene intercambiando filas y columnas:

$$A=(a_{ij}) \Rightarrow A^T=(a_{ji})$$

Una matriz cuadrada A de dimensión $n \times n$ es regular si existe otra matriz B de la misma dimensión que cumpla que AB = BA = 1. Si existe, la matriz B se denomina inversa de A, y se denota por A^{-1}.

Normalmente, usaremos vectores fila y vectores columna para representar puntos y vectores, y matrices cuadradas para representar transformaciones lineales. Las dimensiones de nuestras matrices serán 3×3 o bien 4×4 casi siempre.

Comencemos expresando matricialmente el efecto sobre un objeto de una traslación de vector $v = (vx, vy, vz)$. Una transformación de este tipo manda un punto (x, y, z) a otro (x', y', z'), donde

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

o, de manera más compacta, $P' = P + v$. Vemos aquí que esta transformación no es, estrictamente hablando, lineal: sus ecuaciones presentan términos independientes. ¿Cómo expresaríamos analíticamente una reflexión especular alrededor del plano XY? Tal reflexión cambiaría el signo de las coordenadas z, dejando las demás intactas. Tales cambios independientes se expresarían por medio de ecuaciones

$$x' = x$$
$$y' = y$$
$$z' = -z$$

que serían, en forma matricial

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

De hecho, *cualquier* transformación lineal puede ponerse en forma matricial. La receta es:

1. escogemos una base e_1, e_2, e_3
1. aplicamos la transformación a los vectores de la misma, obteniendo $f_1 = Te_1$, $f_2 = Te_2$, $f_3 = Te_3$
2. ponemos los vectores así obtenidos como columnas de la matriz deseada.

Otra transformación usual es la homotecia o escalado por un factor l. Naturalmente, si $0 < l < 1$, tenemos una contracción, mientras que si $l > 1$ tenemos una dilatación. Cuando $l = -1$, la denominamos reflexión respecto al origen. Cada coordenada de un punto queda multiplicada por l, de modo que

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

La *rotación* constituye un caso más interesante. Queda determinada por el eje de rotación y un ángulo. La rotación de ángulo f respecto al eje z puede calcularse usando la receta anterior:

1. Sea (1, 0, 0), (0, 1, 0), (0, 0, 1) la base.
2. rotémosla, obteniendo los vectores (*cos* f*, sin* f, 0), (- *sin* f*, cos* f, 0), (0, 0, 1)
3. construyamos una matriz con esas tres columnas

$$R_\phi = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Podemos comprobar que la transformación de coordenadas es la correcta:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Si rotamos un ángulo q alrededor del eje *x*, obtenemos una matriz distinta:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Y si aplicamos ambas rotaciones en sucesión, obtenemos

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

La composición de ambas rotaciones la proporciona la matriz producto, que es, en este caso:

$$R = R_x R_z = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \cos\phi \sin\phi & \cos\phi \cos\phi & -\sin\phi \\ \sin\phi \sin\phi & \cos\phi \sin\phi & \cos\phi \end{pmatrix}$$

Esto no tiene el aspecto de ninguna de las rotaciones anteriores, pero es una de ellas, alrededor de algún eje. ¿Cómo podemos saberlo? Porque la matriz resultante es ortogonal:

$$R^t R = R R^t = 1$$

donde 1denota la matriz identidad, que tiene ceros fuera de la diagonal principal y unos en ella. Una matriz ortogonal representa una transformación que preserva las longitudes(y,consecuentemente, los ángulos y los productos escalares). Un cizallaje no preserva las distancias. Como ejemplo de uno, la matriz

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & S_x \\ 0 & 0 & S_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

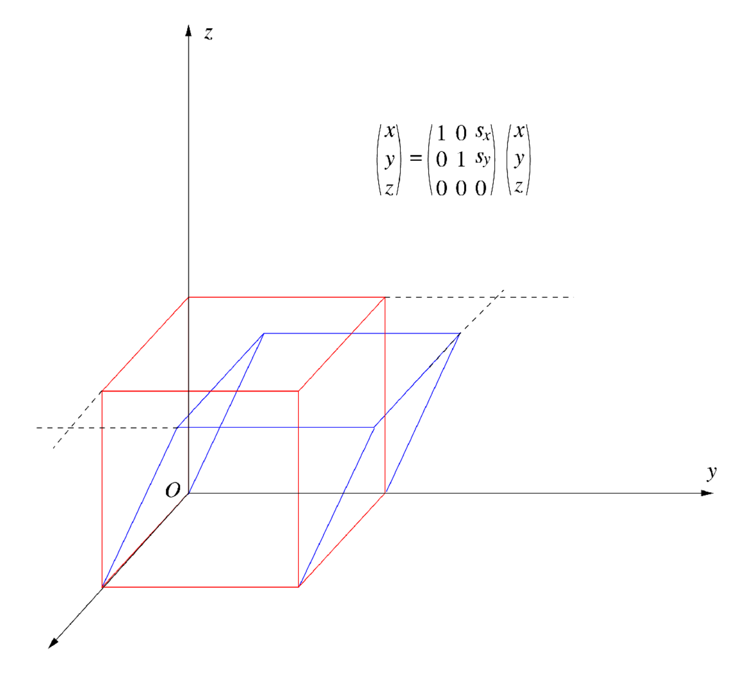


Figura 8. *Cizallaje de un cubo.*

Esto tiene como efecto desplazar las «capas superiores» de un cubo de una manera parecida a como ocurriría con un mazo de cartas, según se muestra en la figura 8.

¿Qué sucede si queremos componer un cizallaje con una traslación, y después aplicar una rotación? La cosa se desorganiza un poquito:

$$X' = R(SX + B) = RSX + RB$$

por culpa de los términos independientes en el producto. En general, obtendremos siempre una transformación del tipo

$$X' = AX + B \quad o \quad \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Ahora podemos apreciar la ventaja de usar coordenadas homogéneas. El espacio afín ordinario puede dotarse de coordenadas con una cuarta componente adicional, de valor unidad, de manera que la ecuación anterior se podrá escribir como

$$X' = AX \quad o \quad \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

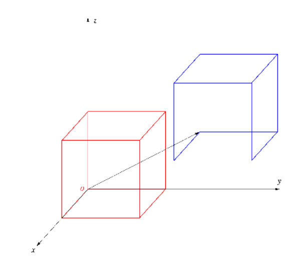
Todas las transformaciones anteriores pueden escribirse así y cualquier transformación afín admite, en coordenadas homogéneas, la forma de una transformación lineal, componiéndose entre ellas por simple producto de matrices.

Sumario proyectivo

Como resumen, veamos la forma que adoptan, en 4–coordenadas proyectivas, algunas de las transformaciones mencionadas anteriormente:

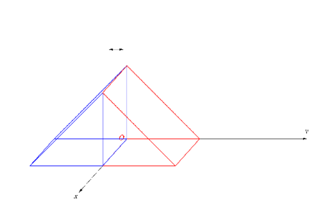
Traslación

$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



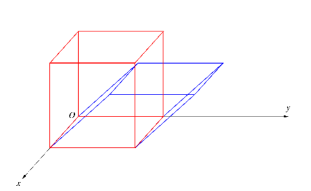
Reflexión

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



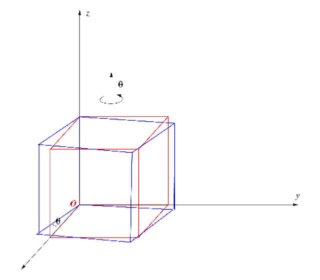
Cizallaje

$$S = \begin{pmatrix} 1 & 0 & S_x & 0 \\ 0 & 1 & S_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



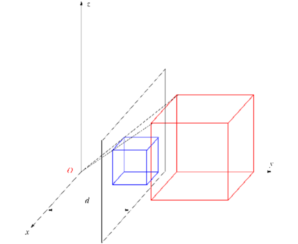
Rotación

$$S=\begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Proyección

$$T=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{d} & 0 & 0 \end{pmatrix}$$



Determinantes

Consideremos una transformación lineal arbitraria, como la representada en la figura 9. La transformación distorsiona la caja azul, convirtiéndola en un paralelepípedo representado en rojo. Hemos hecho coincidir las aristas de la caja con los vectores de la base canónica e1, e2, e3, de modo que las aristas azules correspondientes se transforman en los vectores f1=Te1, f2=Te2, f3=Te3 que son aristas que definen el paralelepípedo rojo.

¿Cuál es el volumen del paralelepípedo? Hay tres hechos obvios

- Si dos vectores fi coinciden, el volumen es nulo.

  V(f1, f2, f3) = 0 si f1 = f2

- Un cizallaje no cambia el volumen (ver figura 8).

  V(f1 + f2, f2, f3) = V(f1, f2, f3)

- La dilatación de un vector dilata de la misma forma el volumen

  V(λf1, f2, f3) = λV(f1, f2, f3)

Las propiedades indicadas son válidas, naturalmente, para cualquier posición funcional de V(·). Estas resultan ser las propiedades que definen de forma única (excepto un factor constante) al determinante de una matriz. Escribimos

$$V(f_1, f_2, f_3) = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} = \sum_{\pi \in S_n} sign(\pi) \prod_{i=1}^{n} f_{i\pi(i)} \qquad (1)$$

La definición general de determinante, que es el tercer término de la fórmula anterior, es

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = +a_{11} a_{22} - a_{12} a_{21}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = +a_{11} a_{22} a_{33} + a_{12} a_{23} a_{31} + a_{13} a_{21} a_{32} - a_{11} a_{23} a_{32} - a_{12} a_{21} a_{33} - a_{13} a_{22} a_{31}$$

Manipulando la definición (1) podemos llegar a la fórmula

$$det(A) = \sum_{i=1}^{n} (-1)^{i-1} det(A_{1i})$$

donde A1i se obtiene de A al suprimir su primera fila y su i–ésima columna. Esta fórmula permite el cálculo por recurrencia de determinantes no muy grandes:

$$\begin{bmatrix} 6 & 3 & 9 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix} = 6 \begin{bmatrix} 3 & 1 \\ 2 & 3 \end{bmatrix} - 3 \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} + 9 \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} = 6(9-2) - 3(6-1) + 9(4-3) = 36$$

Para órdenes mayores, esto es inútil, y resulta preferible usar las propiedades que definen el determinante, a saber: podemos añadir a cualquier fila una combinación lineal de las demás. Esto permite ir haciendo ceros un una columna de forma que el cálculo se reduce a un determinante de orden inferior en virtud del desarrollo anterior det(A). Pero hay una forma mucho más rápida de aplicar esta receta; la explicaremos en la sección siguiente.

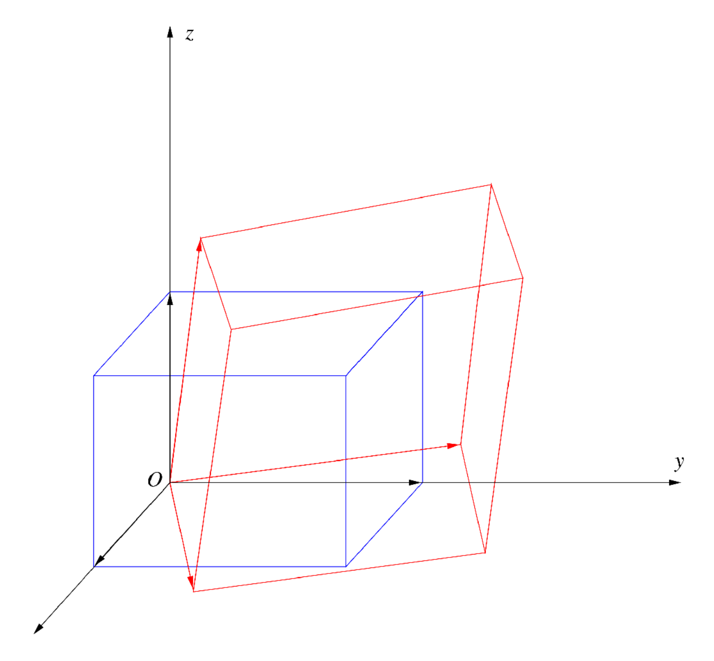


Figura 9. *Determinante como medida de volumen.*

Álgebra lineal numérica y resolución de ecuaciones

En la sección 5, observamos que algunas tareas tienen bastante coste computacional. Nos gustaría aliviar ese coste para que nuestros programas no se perpetúen en bucles interminables. Necesitamos a menudo

- invertir una matriz
- calcular un determinante
- resolver un sistema de ecuaciones lineales
- encontrar una base ortogonal/ortonormal de un subespacio
- encontrar los ceros de un polinomio
- resolver ecuaciones y sistemas no lineales
- ajustar una curva o función a un conjunto de puntos

La mayoría de estas tareas son el tema del *álgebra lineal numérica*; en la cual, no nos importan las propiedades algebraicas de las operaciones lineales, sino solamente encontrar una solución que sea correcta de la forma más rápida posible. Las tareas de la lista anterior en las que no intervienen matrices son el objeto del análisis numérico en general. De

nuevo, tampoco nos interesan las propiedades matemáticas elegantes, sino resolver las cosas, rápido y con precisión suficiente.

La descomposición LU

Llamamos así a una forma embellecida de lo que usualmente se denomina *eliminación gaussiana*: el proceso de hacer ceros en una matriz combinando filas linealmente. La idea nos lleva, al final,a una matriz en forma triangular superior (la parte U de la descomposición). Si además llevamos cuenta de las operaciones realizadas, aplicándoselas a una flamante matriz unidad, obtendremos una matriz triangular inferior (la parte L del monstruo), y, mágicamente, llegaremos a algo como esto:

$$A = \begin{pmatrix} 1 & 6 & 4 \\ 2 & 3 & 5 \\ 8 & 2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 6 & 4 \\ 0 & -9 & -3 \\ 0 & -46 & -29 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 6 & 4 \\ 0 & -9 & -3 \\ 0 & 0 & \frac{-41}{3} \end{pmatrix} = U$$

Registrando los multiplicadores en una matriz unidad nuevecita:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 8 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 8 & \frac{46}{9} & 1 \end{pmatrix} = L$$

y, ¡magia!

$$LU = A$$

El procedimiento puede realizarse in situ (sobre la propia matriz original). Cualquier paquete respetable de álgebra lineal numérica contiene una función estándar que proporciona, por lo común, las partes L y U combinadas en una sola matriz, y una permutación de filas,que se usa para pivotar por razones de estabilidad numérica. así, la descomposición LU se convierte, de forma «oficial»,en

$$PA = LU$$

donde P es una matriz de permutación cuyo efecto es permutar las filas de A. Por ejemplo, si alimentamos a la rutina LINPACK de descomposición LU con

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 4 & -1 \\ 2 & 1 & 5 \end{pmatrix}$$

obtenemos

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad L = \begin{pmatrix} 1.00 & 0.00 & 0.00 \\ 0.50 & 1.00 & 0.00 \\ 0.50 & 0.42 & 1.00 \end{pmatrix} \quad U = \begin{pmatrix} 2.00 & 1.00 & 5.00 \\ 0.00 & 3.50 & -3.50 \\ 0.00 & 0.00 & 2.00 \end{pmatrix}$$

Esta descomposición se usa, principalmente, en tres problemas.

Cálculo de determinantes

Obtenemos

$$\det(A) = \det(P)\det(L)\det(U)$$

Una matriz de permutación como P tiene determinante ± 1 según la paridad de la permutación. La matriz triangular inferior tiene obviamente det(L)=1; y obtenemos, entonces, que, salvo un signo, det(A) es el producto de los elementos de la diagonal principal de U. Esto nos permite el cálculo de determinantes en tiempo O(n3), que es la complejidad usual de la descomposición LU.

Resolución de sistemas de ecuaciones lineales

Supongamos que nos dan un sistema de ecuaciones lineales como éste

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a2_nx_n = b_2$$

$$\cdots$$

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

que escribiremos en forma matricial

$$AX = B \quad o \quad \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ddots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Podemos usar ventajosamente la descomposición LU:

$$AX = P_{-1}LUX = B \qquad \text{o bien} \quad LUX = PB \qquad (2)$$

Salvo una permutación de los términos independientes, podemos resolver la ecuación 2 por sustitución progresiva y regresiva, ya que de

$$\begin{pmatrix} 1 & 0 & \ldots & 0 \\ l_{21} & 1 & \ldots & 0 \\ \ldots & \ldots & \ddots & \ldots \\ l_{n1} & l_{m2} & \ldots & l_{mn} \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

obtenemos

$$w_1 = b_1 \qquad (3)$$

$$w_2 = b_2 - l_{21}w_1$$

$$w_3 = b_3 - l_{31}w_1 - l_{32}w_2$$

y, en general

$$w_k = b_k - \sum_{j=1}^{k-1} l_{kj} w_j$$

Una vez que conocemos los w_i, usamos el mismo truco hacia atrás con U. De

$$\begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \ldots & \ldots & \ddots & \ldots \\ 0 & 0 & \ldots & u_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

escribimos

$$x_n = \left(\frac{1}{u_{nn}}\right) w_n \qquad (4)$$

$$x_{n-1} = \left(\frac{1}{u_{n-1n-1}}\right)\left(w_{n-1} - u_{n-1,n}\, x_n\right)$$

$$x_{n-2} = \left(\frac{1}{u_{n-2n-2}}\right)\left(w_{n-2} - u_{n-2,n}\, x_n - u_{n-2,n-1}\, x_{n-1}\right)$$

y, en general

$$x_k = \left(\frac{1}{u_k}\right)\left(x_k - \sum_{j=1}^{k+1} l_{kj}\, w_j\right)$$

y el sistema está resuelto.

Observemos que si hay que resolver un sistema lineal con varios juegos de términos independientes, la descomposición LU puede reutilizarse, siendo las sustituciones hacia delante y hacia atrás procesos de coste $O(n^2)$ solamente. Esto tiene una aplicación directa en

Inversión de matrices

Aplicando los procesos 3 y 4 a las columnas de la matriz identidad, obtenemos las columnas de la inversa de. Este es,en la práctica,el procedimiento de inversión más efectivo para matrices densas.

Resolución de ecuaciones

¿Qué hacemos si necesitamos resolver una ecuación no lineal? Pongámonos en el caso siguiente: hemos modelado una superficie por medio de una función paramétrica P(u, v) que nos da el vector de posición de un punto genérico de la mismo como función de los parámetros u, v. Y ahora precisamos encontrar la intersección de la superficie con el eje z.

La intersección puede encontrarse resolviendo el sistema

$$0 = Px(u, v)$$

$$0 = Py(u, v)$$

$$z = Pz(u, v)$$

para u, v, z. Pensemos, por ejemplo, que P es una superficie NURBS. ¿Qué hacemos? En primer lugar, pongamos el sistema en la forma

$$f(u,v,z)=0 \quad con \quad f(u,v,z)=\begin{pmatrix} P_x(u,v) \\ P_y(u,v) \\ P_z(u,v)-z \end{pmatrix}$$

En general, este problema es difícil, y tiene que resolverse por un algoritmo numérico (iterativo). Probemos el método de Newton. Empezamos con un valor inicial de *(u, v, z)* = (u_o, v_o, z_o), y lo refinamos repetidamente aplicándole la fórmula

$$(u_{i+1}, v_{i+1}, z_{i+1})=(u_i, v_i, z_i)-Df^{-1}(u_i, v_i, z_i)\, f(u_i, v_i, z_i)$$

Aquí $Df^{-1}(u_i, v_i, z_i)$ es la matriz de derivadas parciales

$$\begin{pmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} & \frac{\partial f_x}{\partial z} \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} & \frac{\partial f_y}{\partial z} \\ \frac{\partial f_z}{\partial x} & \frac{\partial f_z}{\partial y} & \frac{\partial f_z}{\partial z} \end{pmatrix}$$

que se conoce como el jacobiano de f , evaluado en *ui, vi, zi*. Si existe la duda, sí: este es el mismo jacobiano que encontraremos posteriormente en la sección ?? Si tenemos suerte, los puntos sucesivos que iremos obteniendo se aproximarán cada vez más a una solución de la ecuación tras unas cuantas iteraciones. Y, si no tenemos suerte...

La convergencia del método de Newton es cuadrática si tenemos la fortuna de comenzar suficientemente cerca de una solución. Esto quiere decir que en cada iteración duplicamos el número de cifras exactas de la solución, *si todo va bien* (lo que, naturalmente, no tiene por qué ocurrir cuando más lo necesitamos).

Geometría diferencial en cuatro palabras

Curvas

Una curva en el espacio viene dada por una representación paramétrica

$$t \rightarrow \phi(t)$$

donde $\phi(t)$ será un punto genérico de la curva, que es barrido al hacer variar t a lo largo de un intervalo (llamado dominio del parámetro). La velocidad a la que circulamos por la curva es la derivada $\phi'(t)$, y la aceleración es $\phi''(t)$. La longitud de un segmento de curva la da la fórmula

$$L(t_{0,} t_1) = \int_{t_1}^{t_0} \|\phi'(t)\| dt$$

Si ponemos $s = L(t_o, t)$, podemos usar s como parámetro. Esto tiene la ventaja de simplificar bastante algunos cálculos. La descomposición

$$\phi'(t) = \frac{d\phi}{dt} = \|\phi'(t)\| T = vT$$

se convierte en

$$\phi'(s) = \frac{d\phi}{ds} = \|\phi'(s)\| T = T$$

al usar como parámetro la longitud de arco. En ella, T es el vector tangente unitario. Como T · T es constante, 2T' · T = 0 y deducimos que T' es ortogonal a T. Por tanto, es un vector normal cuyo módulo mide la velocidad de rotación de T, la cual es inversamente proporcional al radio de curvatura. Llamamos a |T'| la curvatura k.

$$\phi''(s) = T' = \kappa N$$

con N unitario normal a la curva. Si ponemos B = T × N, es fácil derivar que la tríada ortonormal (T, N, B), como función de la longitud de arco s, satisface las ecuaciones.

T' = kN

N' = - kT - tB

B' = tN

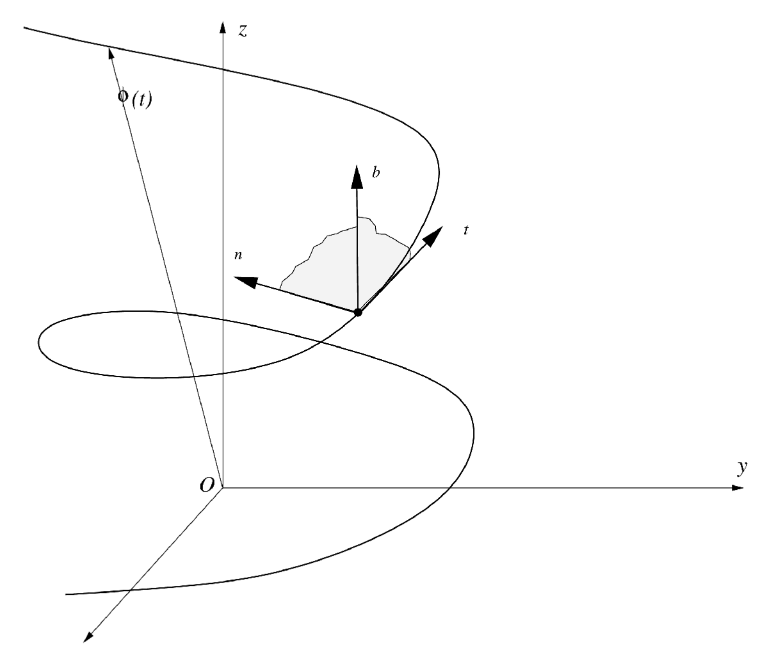


Figura 10. *Triedro móvil de Frenet.*

donde t es una función que mide cuánto dista la curva de ser plana: la torsión. Llamamos a B el vector binormal, y las ecuaciones ?? son las *fórmulas de Frenet-Serret*.

Sólo para físicos: sin duda, se habrán dado cuenta ya de que

$T' = D \times T$

$N' = D \times N$

$B' = D \times B$

si D = kB - tT. ¿Sería mejor decir D = Ω?

Superficies

En la sección 9 veremos que, además de por mallas triangulares o poligonales, podemos modelar superficies empleando funciones más complicadas,como polinomios bicúbicos. En general,podemos describir una superficie por un *parche coordenado*; una función de valor vectorial con dos parámetros (*u*,*v*) que recorre lo puntos de la superficie a medida que *u* y *v* van barriendo su dominio de valores, según vemos en la figura 11. Resulta importante saber cómo realizar cálculos con esta representación.

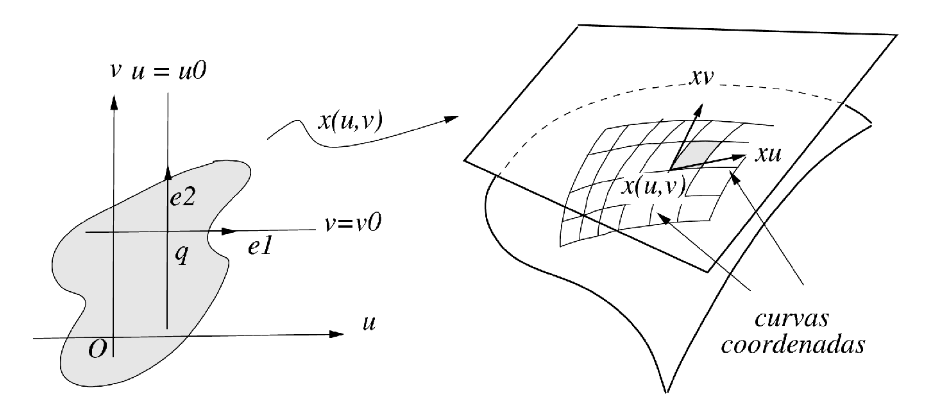


Figura 11. *Superficie Paramétrica.*

En la figura se ve fácilmente que, al mantener fijo *v* y hacer variar *u*, obtenemos una familia de curvas (una curva por cada *v* fijo); y, cambiando los papeles de las dos coordenadas, se obtiene otra familia parametrizada por *u*. Se obtiene, pues, una red de curvas coordenadas que cubren todo el parche. El vector *x(u,v)* es una función de dos variables. Derivando parcialmente obtenemos los dos *vectores tangentes*.

$$x_u = \frac{\partial x}{\partial u}$$

$$x_v = \frac{\partial x}{\partial v}$$

que, junto con el punto x si $b_o = x(u_{o,} v_o)$, definen el plano tangente a la superficie en x_o. La normal en x_o se obtiene la forma habitual

$$N = x_u \times x_v$$

y se puede normalizar si es necesario.

¿Qué sucede con curvas que no son curvas coordenadas? Supongamos que tenemos una curva t→ϕ(t). en forma paramétrica. Si yace en la superficie,tiene que poder escribirse

$$f(t) = x(u(t), v(t))$$

El vector tangente a la curva en cualquier punto viene dado por la derivada:

$$\phi'(t) = \frac{\partial x}{\partial u} u'(t) + \frac{\partial x}{\partial v} v'(t) = x_u u'(t) + v'(t)$$

Vemos, pues, que los vectores tangentes x_u,x_v juegan un papel importante: generan el espacio de todos los vectores tangentes a la superficieen x_o; forman, pues, una base del plano tangente. Pero hay más: observemos el rombo sombreado de la figura 11. Si ignoramos la curvatura, el rombo es la imagen de un rectángulo coordenado de área unidad. ¿Cuál es el área de este elemento de superficie? Conocemos ya muchas formas de expresarla. Por ejemplo:

$$\acute{A}rea = \|x_u \times x_v\|$$

Esta cantidad mide el factor por el cual se multiplica el área al pasar del dominio coordenado *(u, v)* al plano tangente. Es el *jacobiano* de la aplicación del dominio *(u, v)* a la superficie.

Supongamos, por ejemplo, que tuviésemos una masa distribuida por la superficie,dada por la función de densidad *m(u,v)* en coordenadas *(u,v)*. La masa total de la superficie sería

$$\int_{u,v \in D} m(u,v)\|x_u \times x_v\| du\, dv$$

una integral doble que nos recuerda que es preciso tomar en consideración el factor de escala (jacobiano) que el plano coordenado experimenta al aplicarse en la superficie.

Los jacobianos aparecen dondequiera que tenemos transformaciones entre espacios de la misma dimensión y queremos calcular medidas (área, volúmenes o longitudes).

La longitud de arco de la curva f(t) puede calcularse como

$$\int \|\phi'(t)\| dt = \int \|x_u u'(t) + x_v v'(t)\| dt = \int \sqrt{(x_u \cdot x_u) u'^2 + 2(x_u \cdot x_v) u' v' + (x_v \cdot x_v) v'^2}$$

La expresión de la raíz es la primera forma fundamental o tensor métrico

$$I(u,v) = Eu^2 + 2\mathrm{F}\, uv + Gv^2$$

donde

$$\begin{pmatrix} E & F \\ F & G \end{pmatrix} = \begin{pmatrix} x_u \cdot x_u & x_u \cdot x_v \\ x_u \cdot x_v & x_v \cdot x_v \end{pmatrix}$$

Con él, podemos realizar medidas en términos de coordenadas de superficie *(u,v)*. Por ejemplo, longitud de arco

$$\int \sqrt{Eu'^2 + 2\mathrm{Fu}' v' + Gv'^2}\, dt$$

o área de superficie

$$\int_D \sqrt{EG - F^2}\, dudv$$

Una vez más, el radical es el jacobiano de la aplicación que lleva *(u, v)* a la superficie intrínseca.

Interpolación y aproximación

Estos dos problemas aparecen frecuentemente en modelado geométrico.

Fórmula de interpolación de Lagrange

Nos dan n + 1 puntos en el espacio P0, P1, ¼, Pn (representados usualmente como valores de una función real, ver figura ??). Precisamos encontrar una función *p* cuyo grafo incida sobre todos los puntos dados. Este es un problema de interpolación. El caballo de batalla en la solución de este teorema se conoce como la *fórmula de interpolación de Lagrange*.

Teorema (Interpolación de Lagrange): Sean n + 1 puntos del plano XY, $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$,..., $P_n = (x_n, y_n)$, todos con diferentes abscisas. Entonces, existe un y sólo un polinomio p de grado n que cumpla

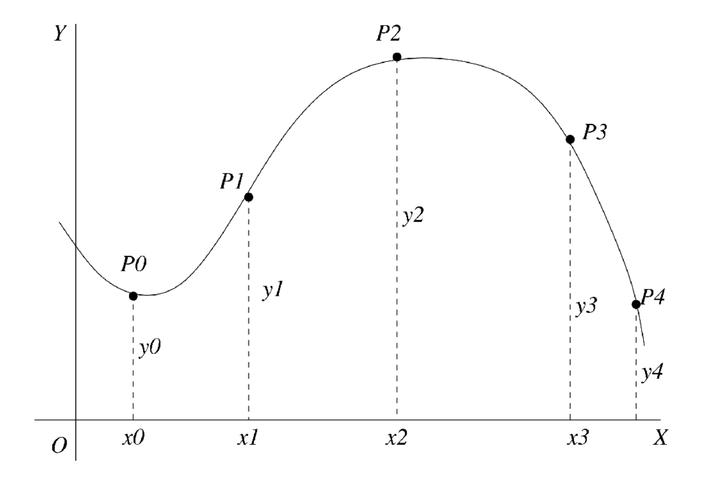
$$p(x_i) = y_i \text{ para } i = 0, 1,\dots, n$$

El polinomio de interpolación de Lagrange puede calcularse de muchas formas, una de las cuales es la fórmula explícita

$$P(x) = \sum_{i=0}^{n} y_k \frac{\omega_k(x)}{\omega_k(k_k)} \qquad (6)$$

donde los factores w son polinomios de grado n dados por

$$\omega_k(x) = \frac{(x - x_0)(x - x_1)\dots(x - x_n)}{(x - x_k)}$$



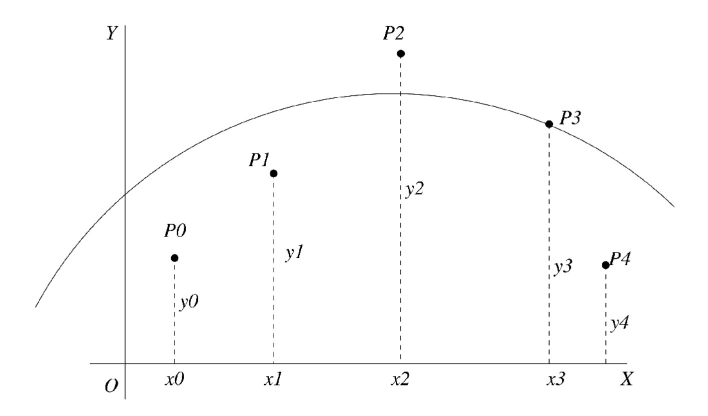


Figura 12. *Interpolación y aproximación de valores puntuales.*

Es obvio que

$$\frac{\omega_k(x)}{\omega_k(k_k)} = \begin{Bmatrix} 1\, si\, i = k \\ 0\, si\, i \neq k \end{Bmatrix}$$

porque ningún par de abscisas coincide. Y esto resuelve nuestro problema.

La fórmula 6, no obstante, no es muy útil en los cálculos. Los coeficientes del polinomio de interpolación se encuentran mejor resolviendo un sistema de ecuaciones lineales, o mediante un uso inteligente de la simetría o la elección hábil de los puntos de muestra (x_i, y_i).

Un ejemplo: queremos encontrar un polinomio cuadrático que interpole los puntos *(- h, y_{-1}), (0, y_0) y (h, y_1)*. En lugar de pasar por el calvario de la ecuación ??, recordemos que el proceso de interpolación es lineal e imaginemos qué pasaría si tuviésemos los conjuntos de valores

$y_{-1} = 1$	$y_0 = 0$	$y_1 = 0$
$y_{-1} = 0$	$y_0 = 1$	$y_1 = 0$
$y_{-1} = 1$	$y_0 = 0$	$y_1 = 1$

En el último caso, buscamos una función cuadrática con ceros en - h y en 0, de manera que tiene que ser

$$q_1(x) = Cx(x + h)$$

con C escogido para que $q_1(-1) = 1$, de modo que $C = 1/(1 - h)$. Por simetría, la primera función q_{-1} será la reflexión especular de ésta, es decir,

$$q_{-1} = 1/(1 - h)\, x(h - x)$$

Y q_0 será una función par por su simetría, con ceros en - 1, 1, de modo que será igual a $(1 - x^2)$ salvo por un factor constante que la normalice para que valga 1 en el origen. ¡Oh!, el factor resulta ser la unidad, de modo que

$$q(x)=y_{-1}\frac{x(x+h)}{1-h}+y_0(1-x^2)+y_1\frac{x(h-x)}{1-h}$$

de forma mucho menos agónica que pasar por la ecuación 6.

Ajuste por mínimos cuadrados

Otro problema distinto es el de aproximación: obtener una curva que pase cerca de los puntos dados, en algún sentido establecido. Se trata de un problema vasto y difícil. Los criterios de cercanía varían mucho según las aplicaciones, y aquí vamos a aproximarnos sólo cuadráticamente (con perdón).

Nos referiremos tan sólo a la forma más primitiva, humilde y, sin embargo, útil de aproximación: el *ajuste por mínimos cuadrados*. Una aplicación corriente de esta técnica es la solución de un problema de interpolación sobredeterminado, en el que tenemos más ecuaciones que incógnitas.

Supongamos, por ejemplo, que nos dan una serie de puntos

$$Pi = (xi, yi)$$

con $i = 1... n$, y sea $n > 4$. Nos obligan a aproximar a ellos una curva que

tiene que definirse por un polinomio cúbico $p(x) = ax^3 + bx^2 + cx + d$. Así que tratamos de hacer mínimo

$$E(a,b,c,d)=\sum_{i=1}^{n}(y_i-ax_i^3-bx_i^2-cx_i-d)^2$$

Esto es, como máximo,una función cuadrática de los parámetros indeterminados que constituyen los coeficientes *a, b, c, d*. Para minimizar, obligamos a que las derivadas parciales de E respecto a ellos sean nulas:

$$\frac{\partial E}{\partial a}=-2\sum_{i=1}^{n}x_i^3(y_i-ax_i^3-bx_i^2-cx_i-d)=0$$
$$\frac{\partial E}{\partial b}=-2\sum_{i=1}^{n}x_i^2(y_i-ax_i^3-bx_i^2-cx_i-d)=0$$
$$\frac{\partial E}{\partial c}=-2\sum_{i=1}^{n}x_i(y_i-ax_i^3-bx_i^2-cx_i-d)=0$$
$$\frac{\partial E}{\partial d}=-2\sum_{i=1}^{n}(y_i-ax_i^3-bx_i^2-cx_i-d)=0$$

Desarrollando y despejando los coeficientes *a, b, c, d,* nos queda un sistema

$$\begin{gathered}\sum x_i^3 y_i=a\sum x_i^6+b\sum x_i^5+c\sum x_i^4+d\sum x_i^3\\ \sum x_i^2 y_i=a\sum x_i^5+b\sum x_i^4+c\sum x_i^3+d\sum x_i^2\\ \sum x_i y_i=a\sum x_i^4+b\sum x_i^3+c\sum x_i^2+d\sum x_i\\ \sum y_i=a\sum x_i^3+b\sum x_i^2+c\sum x_i+dn\end{gathered}$$

o, si lo preferimos así

$$\begin{pmatrix}\sum x_i^6 & \sum x_i^5 & \sum x_i^4 & \sum x_i^3\\ um\, x_i^5 & \sum x_i^4 & \sum x_i^3 & \sum x_i^2\\ \sum x_i^4 & \sum x_i^3 & \sum x_i^2 & \sum x_i\\ um\, x_i^3 & \sum x_i^2 & \sum x_i & n\end{pmatrix}\cdot\begin{pmatrix}a\\ b\\ c\\ d\end{pmatrix}=\begin{pmatrix}x_i^3 y_i\\ x_i^2 y_i\\ x_i y_i\\ y_i\end{pmatrix}$$

Los coeficientes *a,b,c,d* pueden hallarse finalmente resolviendo la expresión anterior.

Modelado de curvas y superficies: splines

Nos dan n + 1 puntos $P_0, P_1, \ldots, P_n$, y desearíamos construir una curva que pase por ellos con la menor *contorsión* posible. Se trata de un problema de *interpolación*. En otras ocasiones, los puntos dados son simplemente *puntos de control*, y queremos que la curva pase cerca de los puntos, o toque algunos de ellos mientras que solamente se acerque a otros. Se trata de un problema de aproximación.

En gráficos por computador, normalmente resolvemos estos problemas usando funciones enlatadas (¿precocinadas?) con juegos definidos de parámetros que es preciso calcular para que la curva cumpla nuestros requisitos. Hay una serie larga de opciones a nuestra disposición:

- Splines naturales
- Interpolantes de Hermite
- Curvas de Bézier
- B-splines
- NURBS

Splines naturales

Un primer enfoque es intentar interpolar por segmentos. El segmento de curva que va de P_i a P_i+1 será una función $p_i(t)$ con t recorriendo el intervalo (0, 1). Las condiciones a imponer son cuatro

$$p_i(0) = P_i$$
$$p_i(1) = P_{i+1}$$
$$p'_i(1) = p'_{i+1}(0)$$
$$p''_i(1) = p''_{i+1}(0)$$

porque cada p_i es un polinomio cúbico. Las dos últimas identidades son condiciones de regularidad, para que la curvatura no tenga «saltos».

El inconveniente de este enfoque es que cada uno de los puntos de control afecta a todos los segmentos, obligando a recalcular todo si cambiamos un simple punto. Y ello implica la resolución de un sistema de ecuaciones para cada tramo.

Interpolantes de Hermite

Una solución similar es la interpolación de Hermite, en que prescribimos las derivadas (tangentes) en los extremos de cada intervalo.

$p_i(0) = P_i$
$p_i(1) = P_i+1$
$p'_i(0) = T_i$
$p'_i(1) = T_i+1$

Esto obliga a especificar las tangentes Ti en los puntos de control, lo que no es siempre posible ni cómodo. La regularidad que se alcanza es menor que con splines naturales; es el precio que hay que pagar por tener control local.

Curvas de Bézier

Esta solución se basa en los llamados polinomios de Bernstein, una familia de polinomios en el intervalo (0, 1) con propiedades muy interesantes de aproximación uniforme. Dada una función P(u) con u Î (0, 1), su n-ésimo polinomio de Bernstein $B_n(P, u)$ viene dado por la fórmula

$$B_n(P,u)=\sum_{k=0}^{n} P(k/n)\, t^k (1-t)^{n-k}$$

Si nos dan n + 1 puntos de control, dibujamos la curva de Bézier asociada a ellos usando la expresión paramétrica

$$B_n(u)=\sum_{k=0}^{n} P_k \binom{n}{k} t^k (1-t)^{n-k} \quad 0\leq u\leq 1$$

Como de costumbre en interpolación y aproximación, las funciones de peso de Bézier constituyen una *partición de la unidad*

$$\sum_{k=0}^{n} P_k \binom{n}{k} t^k (1-t)^{n-k}=1$$

lo que, con su positividad,hace que el proceso sea convexo:la curva de Bézier yace dentro del cierre convexo de los puntos de control. En la figura 13 pueden verse gráficas de algunas funciones base de Bézier.

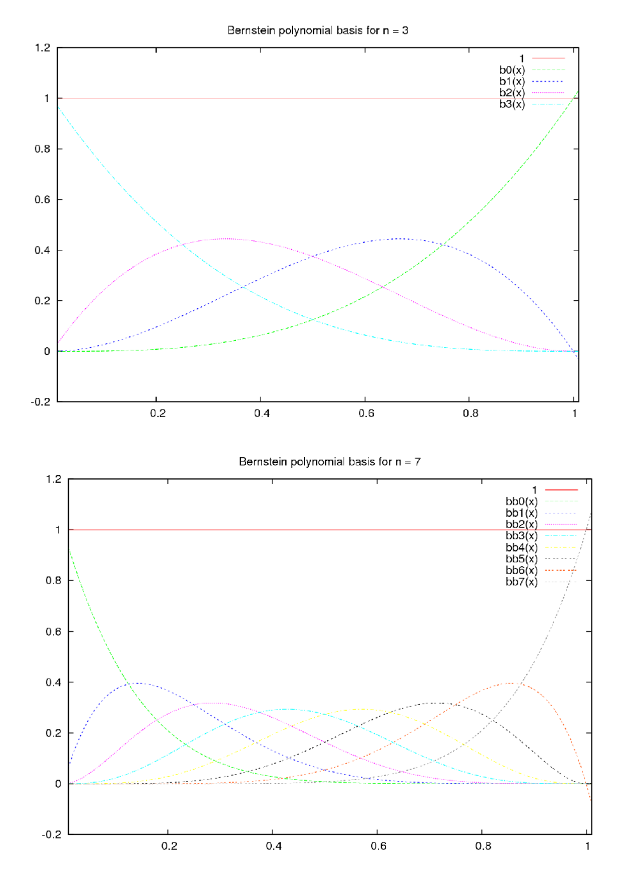


Figura 13. *Base de Bézier para n=3 y n=7.*

En la aplicación más corriente, las curvas de Bézier aparecen como parches con cuatro puntos de control. Es un ejercicio sencillo (¡hágase!) comprobar que P_0P_1 y P_2P_3 determinan la tangente (la velocidad, de hecho) en los extremos,y que la curva pasa por los puntos de control finalesP_0 y P_3, como se ve en la figura 15.

B-splines

Un problema de las técnicas usuales de interpolación es que la curva completa depende de cada punto de control; el cambio de uno de ellos obliga a calcular de nuevo (y dibujar también) toda la solución.

Los B-splines son un conjunto especial de interpolantes carentes de esa deficiencia. Supongamos que tenemos n + 1 puntos de control P_o, P_1, ... , P_n. La curva solución será

$$P(u)=\sum_{k=0}^{n} P_k B_{k,d}(u)$$

donde las funciones $B_{k,d}$ son B–splines de orden d cuyo dominio de definición y propiedades vamos a definira continuación.

Las funciones $B_{k,d}$ se construirán en el intervalo umin, umax dividido en *n* + *d* subintervalos por puntos llamados nodos, de modo que

$$\{u_{min}=u_{0,}u_{1,}u_{,}\ldots,u_{k+d}=u_{max}\}$$

será el dominio de $B_{k,d}$. Cada una de estas funciones será nula excepto en (u_k, u_{k+d}) (lo que abarca d subintervalos del dominio), donde coincidirá con un polinomio de grado d - 1 con valores positivos. Se deduce, entonces, que cada valor *P(u)* será influido por d puntos de control. ¿Cómo se construyen estas funciones mágicas? Por la fórmula de *recursión de Cox-DeBoor*:

$$B_{k,1}=\left\{\begin{matrix} 1 & si\,u_k\leq u \\ 0 & en\,otro\,caso \end{matrix}\right\}$$

$$B_{k,d}=\frac{u-u_k}{u_{k+d-1}-u_k}B_{k,d-1}+\frac{u_{k+d}-u}{u_{k+d}-u_{k+1}}B_{k+1,d-1}$$

A pesar de su apariencia imponente, la recurrencia es bastante sencilla de aplicar. Podemos ver qué sale de ella en la figura 14. Hemos escogido un conjunto de nodos {0, 2, 3, 6, 7, 8, 11, 13} y construido a mano (bueno, casi) los B–splines de ese conjunto de nodos, hasta el cuarto orden (es decir, hasta los polinomios cúbicos). En las gráficas, las propiedades de los B–splines son bastante evidentes.

En la subfigura (e) se verifica otra propiedad interesante: los B–splines de un grado fijo cualquiera proporcionan una *partición de la unidad,* es decir, son funciones positivas de suma unidad en todo punto. Esta propiedad es crucial, haciendo el proceso de interpolación convexo. En la figura 16 vemos la curva B-spline de grado tres con puntos de control

P0 = (0, 0)　　　P1 = (1, 1)
P2 = (0, 2)　　　P3 = (1, 3)
P4 = (0, 4)

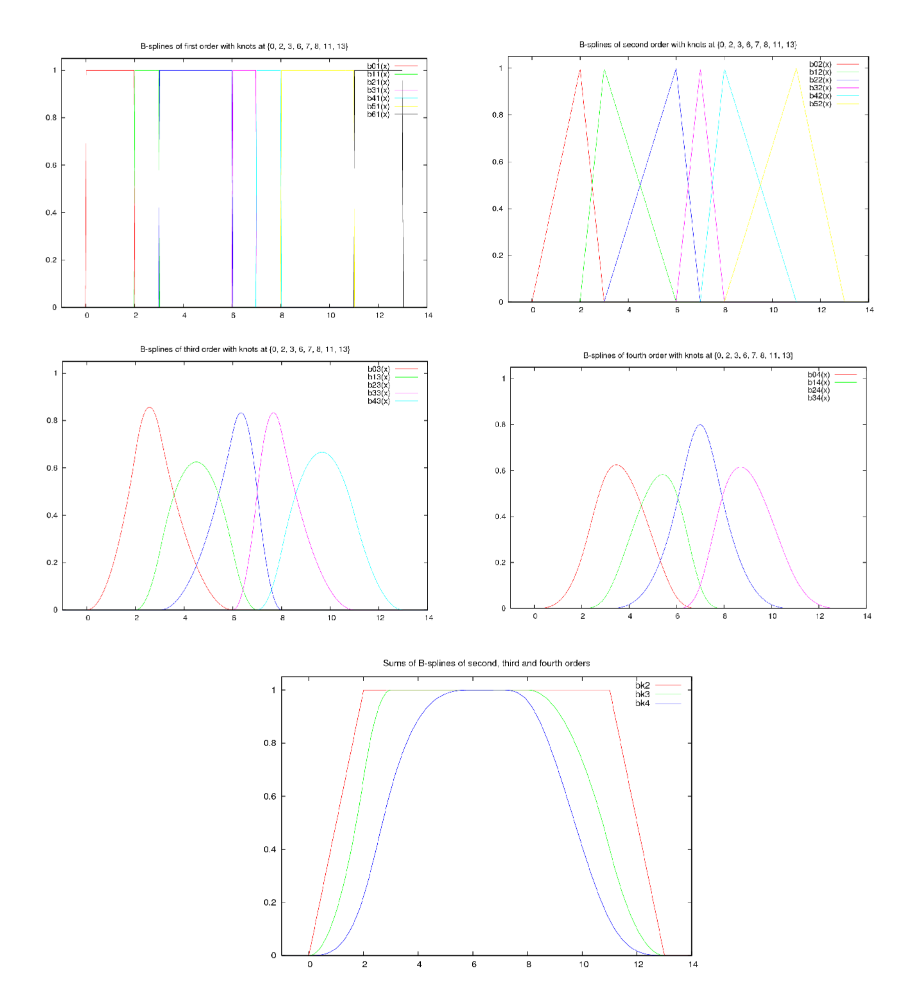


Figura 14. *Gráficas de las funciones de base B-Spline para el conjunto de nodos {0,2,3,6,7,8,11,13}. En la gráfica inferior (Suma de los anteriores), se verifica la propiedad de partición de la unidad.*

Esta vez, hemos escogido nodos uniformemente espaciados. Podemos observar que la curva está contenida en el cierre convexo de los puntos de control, lo que es una consecuencia de la positividad de la base de B-splines, y de que su suma es la unidad.

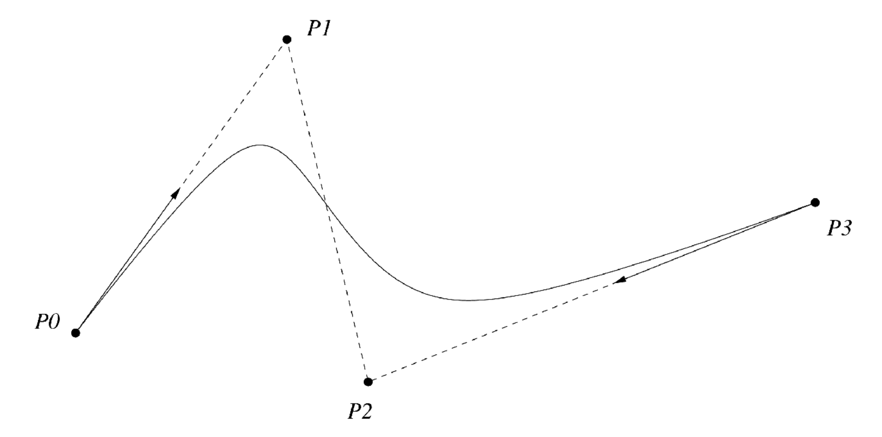


Figura 15. *Curva de Bézier ajustada a los puntos de control extremos y con tangentes que apuntan a los puntos de control intermedios.*

NURBS

Los B–splines pueden aplicarse sin cambios si trabajamos con coordenadas proyectivas, y obtenemos así NURBS (siglas de Non–Uniform Rational B–Splines). La fórmula resultante

$$P(u)=\frac{\sum_{k=0}^{n} P_k w_k B_{k,d}(u)}{\sum_{k=0}^{n} w_k B_{k,d}(u)}$$

en que el denominador puede tomarse como la componente homogénea de los puntos de control en coordenadas proyectivas, incorpora también factores de peso w_k, lo que tiene como consecuencia que las funciones de base son ahora funciones racionales del parámetro.

La ventaja principal de esto es su *invariancia proyectiva*: un B-spline no racional, transformado por una proyección, deja de ser normalmente un B–spline. Dicho de otro modo, el B-spline de los puntos de control transformados no coincide con el transformado del B-spline original. Sin embargo, esa coincidencia sí se da en los B-splines racionales, de forma que para transformarlos basta aplicar la transformación a los puntos de control.

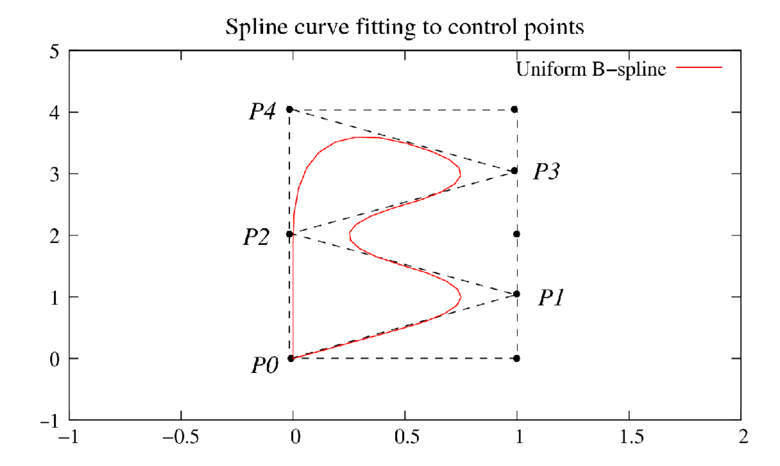


Figura 16. *Aproximación a cinco puntos de control por B-spline cúbico uniforme.*

Splines para superficies paramétricas

Cuando trabajamos con superficies, podemos aplicar las mismas técnicas que con curvas, palabra por palabra. Ahora que tenemos dos parámetros, *u, v*, precisamos como base el producto tensorial de las bases spline en cada uno de ellos. Por ejemplo, podemos interpolar un parche de superficie a través de una red de puntos de control Pij por medio de la fórmula

$$\sum_{ij} P_{ij} B_{i,d}(u) B_{j,d}(v)$$

La propiedad de partición de la unidad sigue siendo válida en la base producto. El mismo truco se aplica a parches de Bézier o NURBS.

Tu cara me suena

Emilio José Molina Cazorla
ej-molina@terra.es ::



La luz y el sonido tienen algo así como una relación de hermandad. La luz es, valga la redundancia, el hermano luminoso, el bonito, el que todo el mundo conoce y en el que todos se fijan. Quién más y quién menos ha oído hablar de fotones, conos y bastoncillos retinas, colores...

Inst... Conceptos básicos

El sonido es el hermano malo, el del Lado Oscuro. Nadie se acuerda de él... excepto cuando falla. Porque, si la imagen es el físico, el sonido es, desde luego, el alma de lo audiovisual. Vamos a intentar darlo a conocer durante las próximas líneas.

¿Qué es el sonido?

Como "hermanos" que son, el sonido y la luz tienen muchas cosas en común, y otras particulares. Mientras que la luz es una onda electromagnética, capaz de propagarse en el vacío y excitar mediante fenómenos fotoquímicos los terminales nerviosos de nuestras retinas, el sonido es una onda de presión, que se propaga mediante procesos de compresión-descompresión del medio en el que se encuentra. Para hacerse una imagen mental, es algo parecido al típico juguete de movimiento continuo (figura 1).

Por ello, en el vacío no se propaga al no haber nada (venga, va, quisquillosos: casi nada) que comprimir, y esto tiene dos graves consecuencias:

- Todas las películas de batallas espaciales son un gran embuste.
- En el espacio, nadie puede oír tus gritos.



Figura 1. *Típico juguete de movimiento continuo.*

Velocidad del sonido, Refracción, Reflexión, Interferencias

Sin embargo, comparte con la luz el que su velocidad cambie según el medio que atraviesa, y los fenómenos de refracción, reflexión e interferencias. En medios poco densos (por ejemplo, el caucho), su velocidad de transmisión será menor que en otros más densos (por ejemplo, el acero).

La velocidad de la luz (casi 300.000 Km/s en el vacío) es inmensamente más alta que la del sonido (340 m/s en el aire a 20ºC). Veremos aquello que provoca un sonido antes de escuchar el sonido que ha provocado. Primero vemos el relámpago y luego escuchamos el trueno. Esto tiene alguna implicación en la creación de animaciones, sobre todo en lo que respecta al lipsync o sincronización labial: si colocamos la voz del personaje a la vez que el movimiento labial, nuestro cerebro se quejará. Sin embargo, si lo desplazamos un par de frames más tarde, se verá (y escuchará) de forma mucho más natural.

El fenómeno de refracción es fácilmente perceptible cuando estamos buceando. Las voces que vienen de fuera (las que vienen de dentro son otro problema que ha de ser tratado con medicación) se escuchan distorsionadas por el cambio de velocidad y de dirección de la onda.

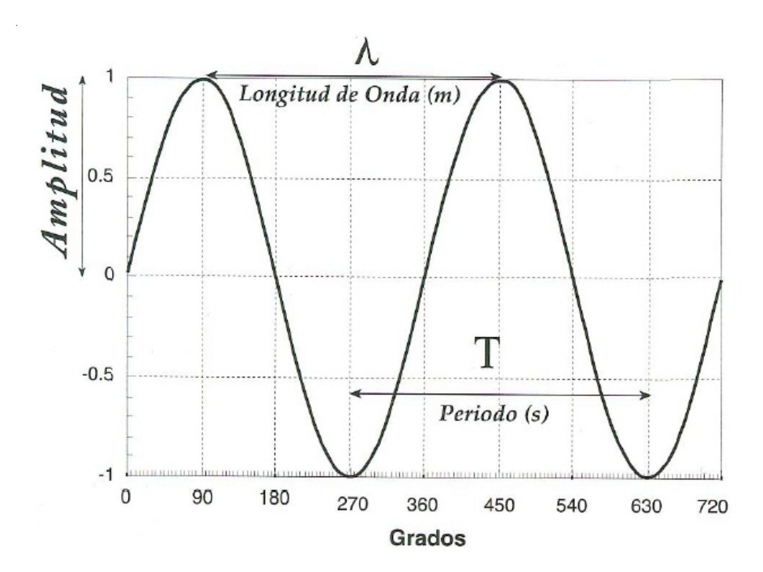


Figura 2. *Onda típica.*

La reflexión, nos es mucho más familiar. Tiene mucho que ver con los coeficientes de absorción de los materiales, y vulgarmente lo conocemos como eco (más adelante entraremos en detalle sobre este aspecto). El reflejo no siempre es idéntico a la onda original; al igual que los objetos reflejan todas las frecuencias de la luz, excepto las que absorben (dando lugar a la percepción de los colores), también absorben más ciertas frecuencias de sonido, provocando efectos curiosos como el de la "carencia de eco" del pato, que en realidad no es más que el producto de la absorción de ciertas frecuencias, reflejando sólo una porción no audible para el oído humano.

A menos que seáis constructores de auditorios y teatros, o queráis buscar la localización idónea de vuestro flamante nuevo equipo de Home Cinema, no es necesario que os preocupéis de este asunto.

El sonido es una onda y, como tal, sufre fenómenos de interferencias. Los sonidos cuyas crestas están en la misma fase, incrementan su intensidad. Los que están en fases contrarias, se atenúan. El primer caso es muy importante de cara al audio digital, donde los sonidos fácilmente "saturan" el tope de sensibilidad de los micrófonos, provocando un resultado "cascado". El segundo es interesante de cara a aplicaciones experimentales de reducción de ruido en lugares cercanos a sitios con alta contaminación acústica, ya que programas en tiempo real producen ondas complementarias que contribuyen a disminuir localmente dicho ruido, cancelando las fases del sonido. También se utiliza en la reducción de ruido en el post-procesado de audio.

Tono, Timbre, Intensidad

Hemos hablado de crestas de ondas. Veamos una imagen de una onda típica (figura 2).

Una imagen está compuesta de diferentes colores con distinta saturación que dibujan formas. Una onda tiene una frecuencia fundamental (inversa a su longitud de onda), más o menos amplias y con unas determinadas frecuencias secundarias.

El tono es la altura del sonido. Cuanto menor sea su frecuencia (y por tanto, más amplia su longitud de onda), más grave será la nota. Al igual que en la luz sólo llegamos a captar frecuencias hasta un nivel de rojos (por encima de los infrarrojos), empezamos a captar sonidos a partir de los 20Hz (un herzio es un ciclo por segundo). Los tonos medios corresponden a frecuencias de unos 300 a 5000Hz (5KHz), y los agudos alcanzan hasta los 20KHz, que sería el límite de la audición humana (en la luz equivaldría al límite entre violeta y ultravioleta).

El timbre es la "forma" de la onda, sus frecuencias secundarias, y es lo que nuestro cerebro utiliza para distinguir la voz de Darth Vader de la de nuestra madre. Bueno, dependiendo de lo mucho que fume. O un piano, un violín o un oboe (por poner un ejemplo más clásico, suponiendo que estén tocando ese tipo de música) de una taladradora (según la pericia del practicante, la diferencia puede radicar en el sutil vibrato de la taladradora).

Podemos ver un ejemplo de las diferencias entre una nota de un piano, un violín, un oboe y un taladro en la Figura 3.

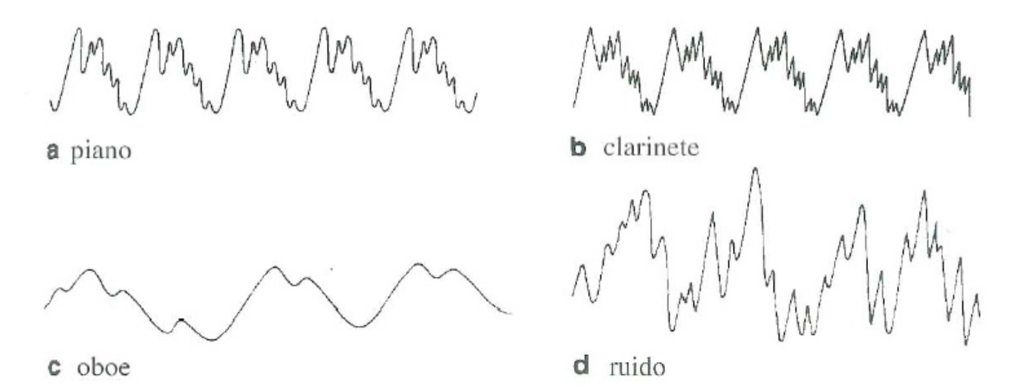


Figura 3. *Diferencias entre una nota de un piano, un violín, un oboe y un taladro.*

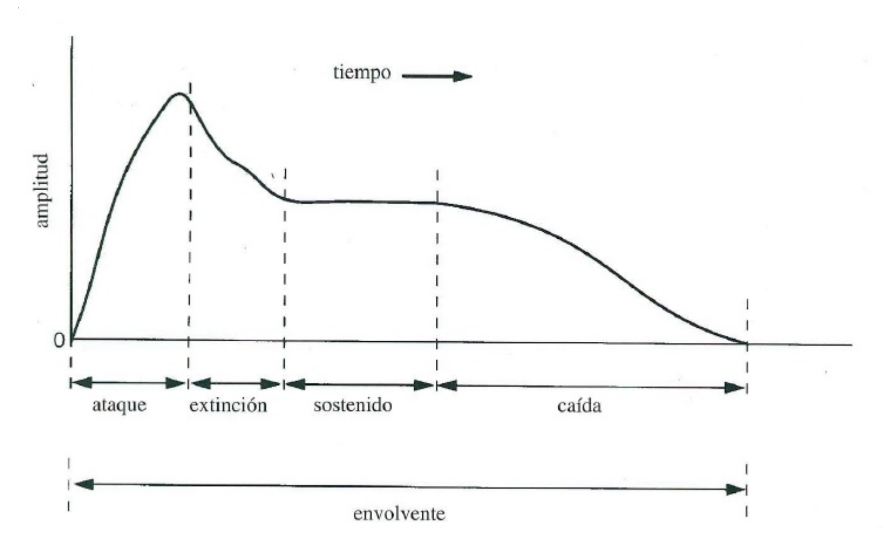


Figura 4. *Envolvente de una onda.*

También es muy importante la forma general de la onda (llamada envolvente), que también utiliza intensivamente nuestro cerebro para catalogar qué tipo de sonido estamos procesando (percusión, voz humana, instrumentos de viento...).

La intensidad de la onda tiene relación con la amplitud de dicha onda, y se traduce en el volumen con el que percibimos un sonido. Recordemos que el sonido es una onda de presión (y como tal, se podría medir en pascales). Sin embargo, el convenio general para medir el sonido es el de la relación de la intensidad del mismo con respecto a la intensidad de un ruido base, y su unidad son los deciBelios. Al igual que los terremotos, sigue una escala logarítmica, donde cada grado superior implica multiplicar la potencia. En la Figura 5 podemos ver las distintas intensidades, en decibelios, de varias fuentes de sonido distintas.

Psicobiología de la percepción acústica

En realidad ya hemos empezado a hablar de percepciones, de lo que el cerebro interpreta o deja de interpretar. Sigamos, pues, con ello.

Antes hablábamos del fenómeno de la reflexión, y decíamos que se conocía como eco. En realidad, el eco no es más que una reflexión que tarda más de 50 milisegundos en llegar a nuestros oídos a partir de la fuente original. A menores tiempos, el cerebro no lo entiende como dos ondas diferentes,

sino como una reverberación de la misma onda, una “cola residual” que, a efectos psicológicos, dota de calidez, profundidad e intensidad a un sonido (sólo hay que pensar en el estudiadísimo efecto de reverberación de las catedrales y los órganos, o en el leve eco que se suele dejar en las baladas). De la misma forma, la “iluminación global” o secundaria, proporciona mayor calidez y “organicidad” a las imágenes que hacen uso de ella, le da mucha más riqueza.

En el mundo sonoro, por tanto, hay que tener muy en cuenta cuál va a ser la reverberación final para conocer el impacto que un sonido tendrá sobre el público. Los estudios de grabación, las salas de cine, y por supuesto los auditorios y teatros han de estudiar a fondo los materiales y su disposición para incrementar o menguar los efectos de reverberación.

En la Figura 6 podemos ver gráficamente cómo Jandro (¡el creador de YafRay!) recibe un haz de ondas sonoras.

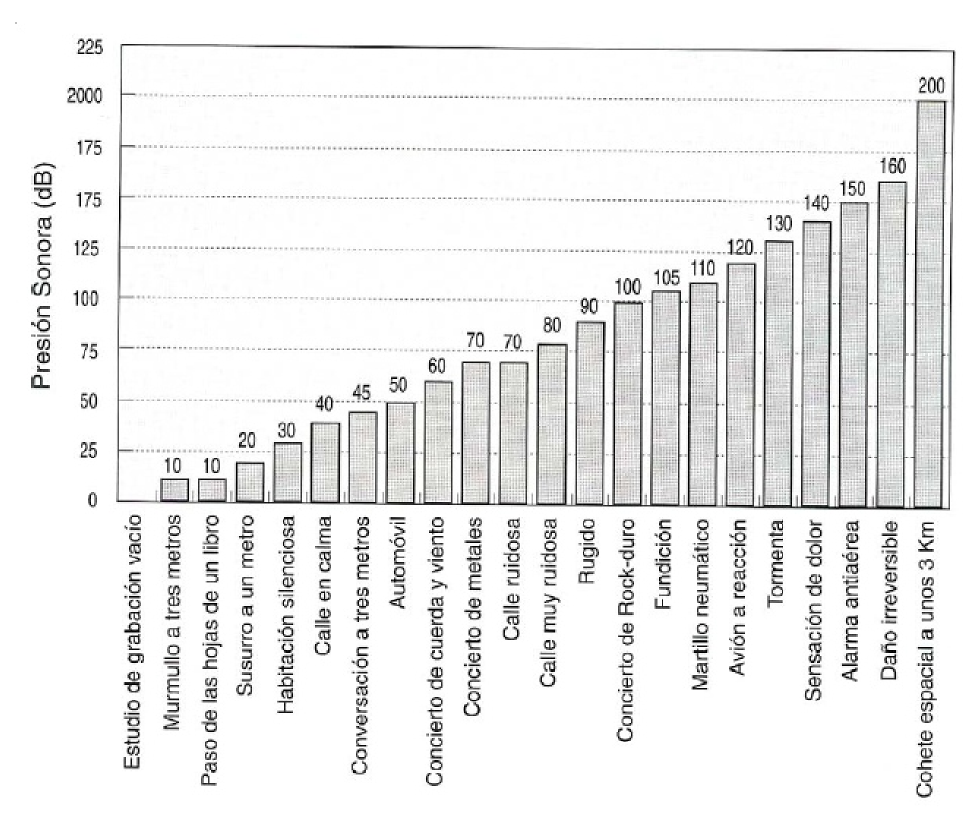


Figura 5. *Fuentes de sonido y sus intensidades.*

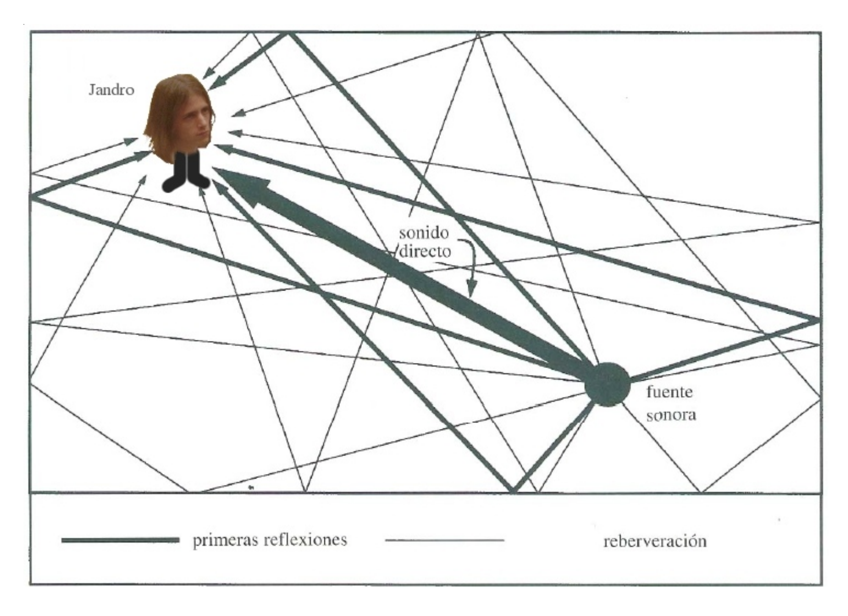


Figura 6. *Jandro recibe un haz de ondas sonoras.*

La resonancia es la vibración “simpática” que se da en algunos objetos cuando la frecuencia fundamental de una onda que suena cerca es múltiplo (también llamado armónico) de su propia frecuencia de sonido. Por ejemplo, ciertas notas tocadas por instrumento de viento son capaces de hacer vibrar las cuerdas de una guitarra, si sus notas son las mismas o frecuencias múltiplos (octavas superiores). O las cuerdas del interior de un piano, cuando se toca una nota (ejercicio práctico: tocar fuerte una nota y, tras soltarla, pulsar el pedal de “sostenido” enseguida; las cuerdas, distensadas, vibrarán cada cual según su resonancia con la nota que hubiéramos pulsado). También nuestras cuerdas vocales tienen resonancia con nuestra propia voz, lo que produce frecuencias secundarias, armónicos, que le dan su timbre característico.

Estéreo, Surround, Efecto Doppler

Tenemos percepción de profundidad en la visión gracias a poseer un par de ojos (a menos que seas alguna especie de tipo raro con casco espacial y no distingas a tu suegra de un piano de cola). Esta estereoscopia permite a nuestro cerebro calcular la profundidad de un objeto a partir de las pequeñas diferencias de ángulo de la imagen recibida por cada ojo individual. Con los oídos pasa tres cuartos de lo mismo: dependiendo de su posición, una onda sonora llegará antes a un oído que a otro.

Si la fuente está más cerca de nuestro oído izquierdo, llegará antes hasta éste, atravesará el cráneo y estimulará más tarde el oído derecho.

Por la forma de nuestro pabellón auditivo, también escucharemos mejor los sonidos que provengan de nuestro frente que de nuestra espalda.

Con ambos factores, somos capaces de descubrir con bastante precisión la localización y lejanía de una fuente sonora a partir de las pequeñas diferencias de tiempo e intensidad.

Con una situación adecuada de fuentes sonoras, podemos conseguir un efecto envolvente (surround), sobre todo si contamos con un altavoz especial para bajos (subwoofer), ya que los altavoces convencionales apenas son capaces de reproducir ciertas frecuencias muy graves.

El efecto Doppler, al igual que se aplica a la luz para descubrir si un emisor se aleja -tiende al rojo- o se acerca -tiende al azul- al observador, muestra el mismo comportamiento en el sonido (al fin y al cabo, en ambos casos hablamos de frecuencias de ondas que se acercan o se alejan). Cuando un sonido se aleja, tiende a escucharse más grave (sus ondas nos llegan “alargadas”), mientras que si se acerca, se escucha más aguda (sus ondas nos llegan “comprimida”). Este efecto nos resulta muy familiar; todos reconocemos el sonido de una ambulancia acercándose y alejándose.

Zonas de percepción

Ni escuchamos las mismas frecuencias por igual, ni todos tenemos los mismos límites de audición. Las razones: milenios de evolución.

Como decía Jack, vayamos por partes. Literalmente. En la Figura 7, podemos ver el sistema auditivo con una serie de partes detalladas.

Recapitulemos: los sonidos son variaciones de presión que se transmiten por un medio no vacío. En el caso humano, este medio suele ser aire. Las ondas son recogidas y concentradas por el pabellón auditivo, que las conduce hasta el tímpano. El tímpano no es más que una membrana (como la de un tambor) que transmite esta vibración a los huesecillos del oído (martillo, yunque y estribo), que a su vez los transmite (ya con mucha menos fuerza) al oído interno (canales semicirculares y caracol). El oído interno se encarga, aparte del importante tema de escuchar, del no menos importante tema de conservar el equilibrio, estático o dinámico (también se encarga de averiguar si nos estamos moviendo). Una vez en el caracol, el sonido que ha sido transmitido por el estribo hace vibrar el líquido que lo contiene. Gracias a su forma en espiral estrechada, cada sección del caracol tiene resonancia con unas determinadas frecuencias. Desde los sonidos graves en la ancha base (a partir de 20 Hz) hasta los más agudos en la punta final (20 Khz). Los enlaces nerviosos de la zona se encargan de

transducir la energía mecánica de la vibración a energía eléctrica, que mandan seguidamente a través del nervio auditivo.

Como podemos ver en la gráfica de respuesta a las frecuencias según la intensidad de la Figura 8, tenemos un muy buen procesamiento de frecuencias medias (ya que es ahí donde se centran los sonidos que la Naturaleza considera más interesantes para nuestra supervivencia: un tigre rugiendo, una suegra acercándose...). No podemos escuchar ni las frecuencias muy bajas ni las muy altas, a menos que suenen a una determinada intensidad.

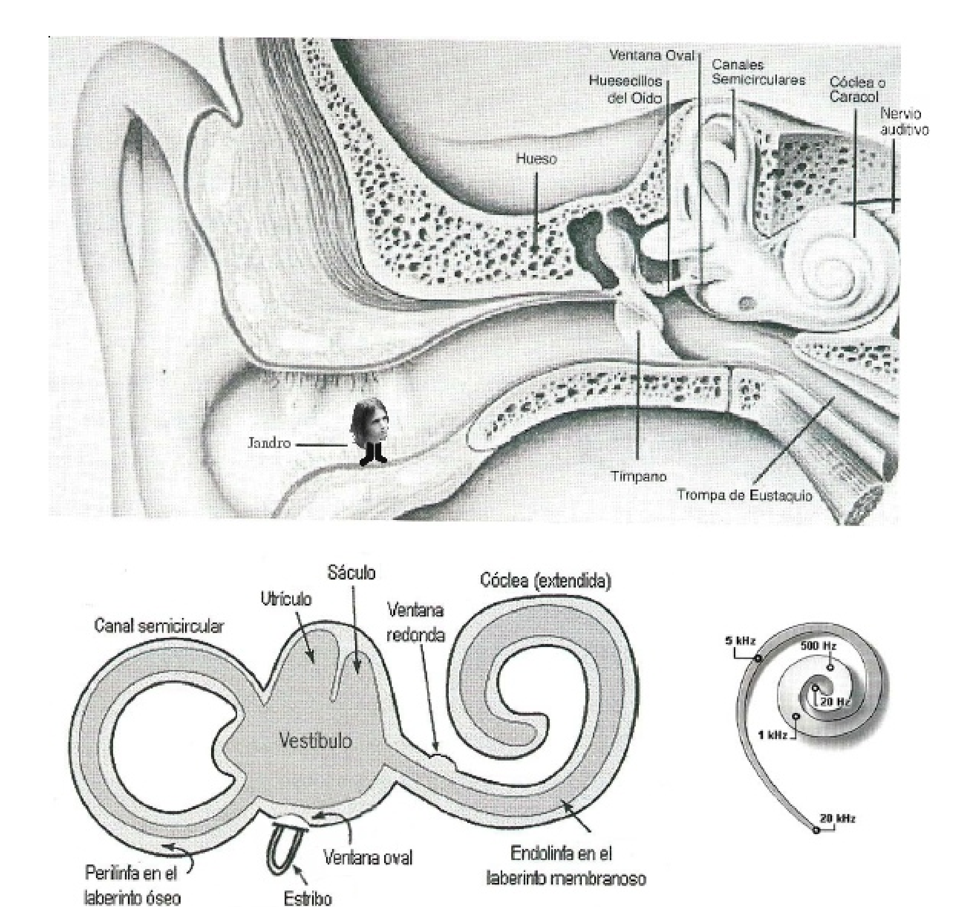


Figura 7. *Sistema auditivo.*

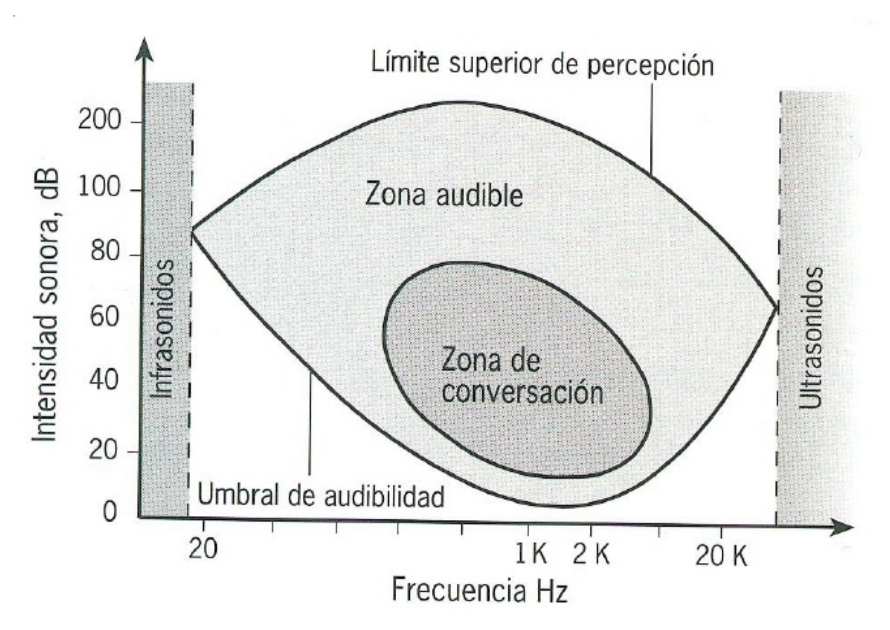


Figura 8. *Respuesta frecuencial.*

Si la intensidad es muy elevada, corremos el riesgo de romper la membrana timpánica, aunque podemos soportar mejor la intensidad en ciertas frecuencias, precisamente por la menor respuesta que tenemos ante las frecuencias graves y agudas.

Además de esto, la pérdida de audición por la edad también es diferente en hombres y mujeres, siendo mucho más acusada en los hombres. Algunos sugieren que es por escuchar continuadamente los gritos de dichas mujeres. Nunca lo sabremos con certeza.

Algún día, todo será digital

Algún día, todo será digital. Por ahora, el audio ya lo es. Con las tecnologías de reducción de ruido como Dolby, la calidad del audio digital es, simplemente, perfecta.

La captura de sonido se realiza mediante micrófonos, que pueden ser de varios tipos según la direccionalidad del sonido que capturan (omnidireccionales, bidireccionales, unidireccionales o cardiodies...) o el método que utilizan para convertir el sonido de presión a electricidad (principalmente, dinámicos y de condensador). Las diferencias son principalmente de cantidad de sonido captado y de capacidad de soportar fuertes intensidades sin que el sonido “sature” y “se rompa”, lo cual suele ser inverso al rango de matices que son

capaces de captar. Dicho de otra forma, las sutiles variaciones de potencial de la corriente eléctrica necesarias para recoger con gran precisión cierto tipo de sonidos, pueden provocar que, ante sonidos intensos, el voltaje se dispare al máximo, destrozando así la forma de onda y convirtiendo toda la información que llevaba ese sonido en un molesto ruido.

Cuando el sonido llega a la tarjeta de ídem, ésta muestrea la señal eléctrica X veces por segundo (frecuencia de muestreo), adjudicando a cada valor de entrada un determinado valor Y de salida dentro de un rango de bits (resolución). Dicho de otro modo, volviendo a las analogías con la imagen, la frecuencia de muestreo sería el tamaño en píxeles de la imagen, y la resolución sería la profundidad de colores de la misma. Cuanta más resolución, mayor número de grados intermedios de intensidad tendrá la onda. Cuanta mayor sea la frecuencia de muestreo, más muestras tendremos por segundo. En conjunto, la onda muestreada será más parecida a la analógica real cuanto mayores sean los valores mencionados.

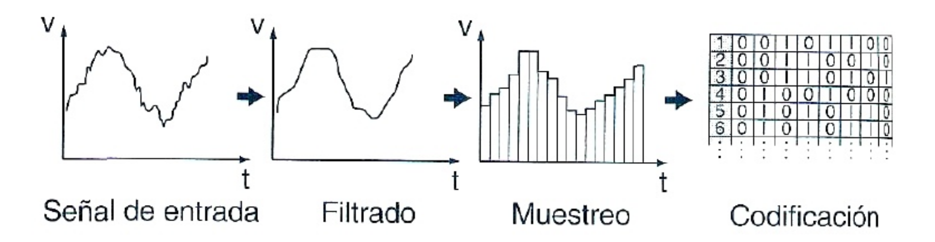


Figura 9. Conversión Analógico-Digital.

Formatos

Después de esto, nuestra onda original se ha convertido en un chorro de números enteros (o incluso reales) en memoria, tantos como canales tuviera el sonido (un canal si es mono, dos si es estéreo, e incluso más para sonidos cuadrafónicos y surround). A la hora de guardar esta información en disco, tenemos dos opciones: guardarlas en formato bruto sin compresión (PCM, Pulse Code Modulation) o utilizar alguno de los formatos de compresión sin pérdida (FLAC) o con pérdida (más populares): MPG-1 Audio Layer 3 (mp3) o Vorbis-OGG (ogg). En contra de lo que normalmente se piensa, el conocido formato wav (WAVeform audio format) es sólo un formato "contenedor" (igual que el AVI para vídeo); el audio que lleve en su interior puede ir en bruto o comprimido con cualquiera de los métodos existentes.

En ambos formatos, mp3 y ogg, la compresión se centra en las frecuencias de la onda inaudibles para el oído humano (siguiendo un modelo psicoacústico), utilizando una tasa de bits variable (distinta cantidad de información por segundo, según lo que esté sonando).

Las diferencias:

Mp3 es un formato patentado por Thompson (por el que cobran importantes sumas en concepto de derechos de códecs en reproductores) y utiliza la Transformada Rápida de Fourier para codificar el sonido.

En cambio, ogg (formato libre) utiliza la Transformada de Cosenos Discreta Modificada y se espera que pronto pueda usar Wavelets, una forma mucho más sofisticada de codificación y análisis, basada en armónicos.

La combinación de alta calidad y formato libre del ogg está produciendo una debacle del formato mp3, si bien es lenta por la enorme extensión de uso del mp3 y el desconocimiento general de la existencia del ogg.

Surfeando en la onda

Una vez tenemos el audio en formato digital, podemos hacer mil virguerías con él utilizando un programa de retoque de audio como el programa libre Audacity `http://audacity.sourceforge.org`. Podemos cambiar la resolución y el muestreo de una onda, pasar de estéreo a mono y viceversa, cambiar el "panning" (la posición del sonido en el estéreo), agregar eco o reverberación, invertir los picos de la onda, invertir la onda al completo, reducir el ruido, acelerar o decelerar la reproducción, variar el tono, modificar su envolvente o su intensidad... Incluso este programa nos ofrece la posibilidad de dibujar la onda, ampliando muchísimo el zoom en ella.

También podemos generar sonidos digitalmente. Para ello tenemos dos opciones: la síntesis aditiva o la síntesis sustractiva. Al igual que en el color tenemos los mapas aditivos (RGB) y sustractivos (CMYK) según si sumamos luces al negro o restamos tintas al blanco, en el timbre podemos "añadirle al silencio" distintas frecuencias generadas mediante funciones sinusoidales o similares, o podemos restarle a una frecuencia dada una serie de frecuencias.

Conclusión y cierre

Según la Wikipedia, la Teoría M es una de las candidatas a convertirse en la Teoría del Todo. Tiene su origen en la Teoría de las Cuerdas, según la cual todas las partículas son en realidad diminutas cuerdas que vibran a cierta frecuencia, y nosotros vivimos en un universo vibrando a cierta frecuencia.

No deja de ser poético el pensamiento de que, después de todo, nuestro Universo, con todo lo que contiene, es sólo una increíble sinfonía. Demasiado bonito como para dejar de lado algo tan cercano a esta realidad.

Visión Estereoscópica

Javier Alonso Albusac Jiménez
Miguel García Corchero
jalbusac@gmail.com · greenbite@gmail.com ::



El mecanismo de visión de los seres humanos es estéreo por naturaleza. Cada uno de los ojos obtiene una imagen del escenario en el que nos encontramos (visión binocular), con un ángulo ligeramente diferente.

Las imágenes adquiridas tendrán bastante contenido visual en común, pero cada una proporcionará información que la otra no tiene, lo que denominamos como **disparidad**.

¿Qué es la visión estereoscópica?

Una vez que los ojos han servido como mecanismo de captación del medio, llega el turno del cerebro, que será el encargado de procesar las imágenes. El cerebro analiza las imágenes generando una imagen en estéreo, producto de la combinación de las similitudes y la detección de las diferencias, pudiendo de esta forma tener la sensación de profundidad, lejanía o cercanía de los objetos que pertenecen al escenario del mundo real. Al proceso de fusión de las imágenes se le conoce como **Estereopsis**.

El procedimiento de Estereopsis puede variar en función de la distancia de los objetos que estamos observando, cuando los objetos son lejanos los ejes ópticos de los ojos permanecen en paralelo, mientras que si de lo contrario se encuentran a una distancia cercana los ejes de los ojos convergen. Se conoce como **fusión** a la suma de alguno de los mecanismos citados anteriormente, más la acomodación y enfoque para ver nítidamente los objetos del escenario.

La capacidad que tienen las personas para apreciar los detalles situados en distintos planos varía con facilidad, esta capacidad está directamente relacionada a la **distancia interocular**. Cuanto mayor es la distancia entre los ojos mayor es la distancia límite a la que se puede seguir apreciando el relieve de los objetos. La distancia límite puede varias entre 60 y cientos de metros entre diferentes personas.

La visión estéreo es necesaria en el día a día de cualquier ser humano, la utilizamos constantemente, sin ella sería prácticamente imposible practicar cualquier deporte, introducir una llave en su cerradura o conducir un automóvil entre otras muchas actividades.

Cómo imitar la visión estereoscópica humana.

Para poder conseguir simular la visión estereoscópica necesitamos de algún dispositivo físico que realice las mismas funciones que los ojos y realicen una captura del entorno, una vez obtenidas las imágenes necesarias se aplicarán una serie de métodos o técnicas para conseguir la sensación de espacio (tarea de la que se encarga el cerebro).

Como dispositivos se suele utilizar una o dos cámaras, tomando un par de fotos del entorno y separadas a una distancia similar a la que se encuentran los ojos. Por tanto tendríamos una fotografía similar a la vista tomada por el ojo izquierdo, y otra a la del ojo derecho.

Existen diferentes métodos para la obtención de imágenes estereoscópicas:

Slide Bar

En este método tan sólo se utiliza una cámara que se encuentra situada sobre una barra de desplazamiento graduada, por lo que moviendo la cámara a lo largo de la barra podremos obtener imágenes desde distintas posiciones similares.

Twin Cameras

Se dispone de dos cámaras situadas en un soporte físico. Las fotografías son tomadas en el mismo instante de tiempo, debe existir por tanto un mecanismo de sincronización entre ambas cámaras.

Beam splitters

En este caso se dispone de una sola cámara SLR(Single Lens Reflex Camera) y un sistema de espejos o prismas situados enfrente de las lentes de la cámara. La colocación correcta de los espejos permite obtener la versión de la parte derecha y la de la izquierda.

Con la aplicación de cualquiera de estos métodos llevamos a cabo la tarea correspondiente a los ojos, la captación de las imágenes, queda por tanto realizar el trabajo del cerebro, consiguiendo así un efecto tridimensional.

Métodos de visualización

Podemos clasificarlos en dos clases. Aquellos en los que no es necesario ninguna herramienta o dispositivo para conseguir la unión de las dos imágenes, conocido como visión libre, y aquellos en los que sí es necesario.

Visión Libre

En la visión libre podemos encontrar dos métodos de visualización, visión paralela y visión cruzada.

Visión paralela

Los ojos mantienen sus ejes ópticos en paralelo, como si mirásemos al infinito. El ojo izquierdo mira la imagen izquierda y el derecho la imagen derecha. Al cabo de un tiempo, deberíamos ser capaces de fusionar las dos imágenes en una única tridimensional. Este método es utilizado en los libros con estereogramas de puntos aleatorios (ojo mágico).

Visión cruzada

Las imágenes son cambiadas de orden, la imagen izquierda se sitúa en la parte derecha, y la derecha en la izquierda, Los ejes ópticos también se cruzan, el ojo derecho mira a la izquierda y el izquierdo a la derecho, para conseguirlo podemos situar uno de nuestros dedos entro los dos ojos y mirarlo.

El método de visión paralela suele utilizarse cuando las imágenes no son superiores a 65 milímetros entre sus centros, en el caso contrario, cuando la distancia entre los centros de las imágenes supera los 65 milímetros (la imagen virtual aparece más pequeña) se utiliza la visión cruzada.

Métodos de visualización con utilización de dispositivos

Proyección

El efecto estéreo se consigue con la utilización de un proyector con dos lentes, o bien dos proyectores normales encargados de mostrar la imagen correspondiente al ojo izquierdo y la del derecho. A cada una de las lentes se le aplica un filtro polarizador asegurando así que cada ojo vea únicamente la imagen que le corresponde.

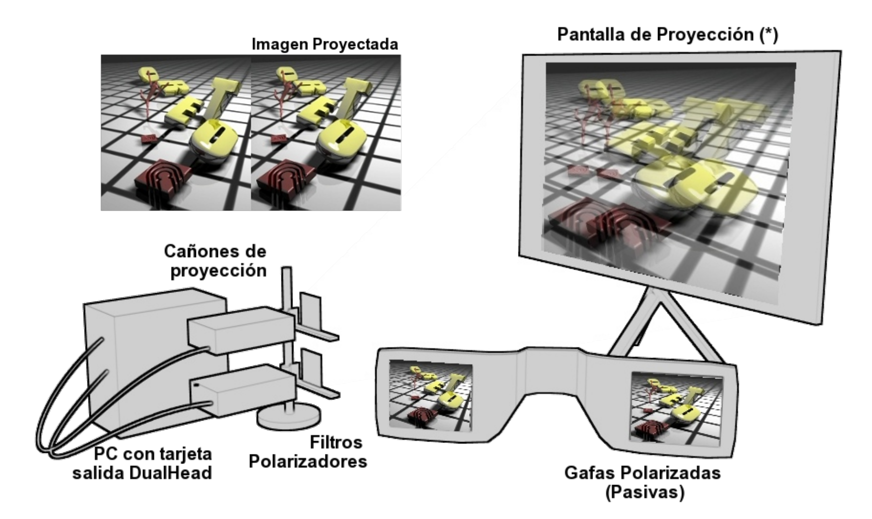


Figura 1. *Esquema de proyección con luz polarizada.*

Las imágenes deben ser proyectadas sobre una pantalla de aluminio distinta al color blanco para evitar la despolarización de la luz proyectada (ver figura).

Anáglifo

Al igual que en los métodos anteriores se trata de engañar al cerebro para aparentar una sensación tridimensional a partir de imágenes planas. Para conseguir que cada ojo se centre únicamente en la imagen que le corresponde se tiñen las imágenes de un determinado color y se utilizan unas gafas apropiadas con filtros de colores.

Se pueden utilizar varías combinaciones de colores, pero la más común consiste en teñir la imagen izquierda de color azul y de rojo la imagen derecha. En la gafas el orden de los colores de los filtros es inverso, el filtro derecho sería azul mientras que el izquierdo sería rojo.

Este método es de los más económicos pero presenta grandes inconvenientes:

- Alteración de los colores originales.
- Perdida de luminosidad.
- Cansancio visual después de un tiempo prolongado.





Figura 2. *Si los filtros polarizadores están alineados (arriba), puede pasar la luz. Si rotamos uno de los filtros 90º, cortan la luz en dos ejes, volviéndose totalmente opacos (abajo).*

Polarización

El sistema es similar a los anáglifos, pero en este caso no se utilizan colores para la separación de las imágenes, si no que se utiliza luz polarizada.

Este sistema se puede construir con dos proyectores, dos filtros que se colocan delante de cada proyector y unas gafas polarizadas. Cada uno de los filtros y cada una de las lentes de las gafas están polarizadas en una dirección y difieren en 90 grados (ver figura 2).

Este método solventa algunos de los inconvenientes que presentaban los anáglifos:

- Respecta en gran medida los colores de la toma original.
- Disminuye el cansancio visual.
- Como inconveniente tenemos que existe pérdida de luminosidad con respecto a la toma original.

Alternativo

En este sistema se utilizan gafas dotadas con obturadores de cristal líquido (denominadas LCS, Liquid Crystal Shutter glasses o LCD, Liquid Crystal Display glasses). Las imágenes son presentadas de forma secuencial, alternando las imágenes para el ojo izquierdo y para el derecho. El parpadeo no es perceptible ya que se utiliza una frecuencia alta de proyección.

Las gafas serán las encargadas de cerrar el obturador del ojo derecho cuando se proyecten imágenes para el ojo izquierdo y viceversa.

Como gran ventaja tienen que la señal sincronizada es enviada a las gafas por rayos infrarrojos y no necesita de ningún cable, pero por contra estas son bastante voluminosas y pesadas.

Casco Estereoscópico

Head Mounted Display (HMD). El casco consta de dos pantallas, una para cada ojo. El casco está conectado a un sistema encargado de enviar las imágenes. Es utilizado principalmente en el área de Realidad Virtual y en Videojuegos.

Posee algunos inconvenientes:

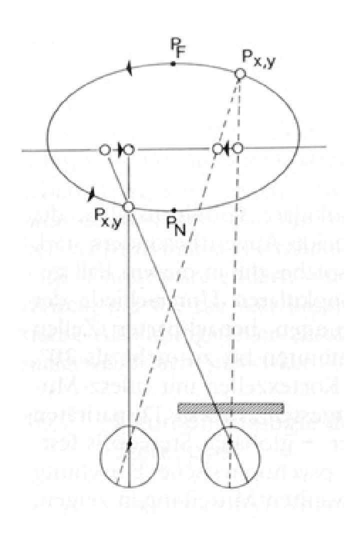
- Elevado coste.
- Voluminosos y pesados.
- Provocan fatiga visual.

Monitores Auto-Stereo

Se trata de unos monitores que aún se encuentran en fase de experimentación y desarrollo. Utilizan microlentes situadas de forma paralela y vertical sobre la pantalla del monitor, generando la desviación de dos o más imágenes. Estos monitores no necesitan el uso de ningún tipo de gafas para obtener la sensación de profundidad.

Efecto Pulfrich

El llamado Efecto Pulfrich fue descubierto por el médico alemán Carl Pulfrich en 1922. La diferencia con respecto a otros sistemas de visualización estéreo es la utilización de una única imagen 2D animada, esto no resta que puedan obtenerse efectos estereoscópicos muy espectaculares filmando con una única cámara en movimiento.



Este fenómeno consiste en la observación de una imagen en movimiento horizontal, el observador debe utilizar un filtro en uno de los ojos. La imagen captada por el ojo en el que se encuentra el filtro posee menor luminosidad que la imagen original provocando que llegue al cerebro con un retardo de centésimas de segundo, el cerebro percibe la imagen con una pequeña diferencia de posición horizontal generando el efecto estereoscópico.

Áreas de Aplicación

Una de los campos donde más se utiliza la visión estereoscópica es en la topografía tomando fotografías aéreas. Para poder llevarla acabo correctamente, el avión debe volar a una altitud constante y realizar fotografías verticales. Suele existir grandes distancias entre una fotografía y la siguiente, dando una gran sensación de relieve que sin la visión estereoscópica no se podría apreciar.

Como ya habíamos mencionado anteriormente la separación media interocular es de 65 mm y es la distancia que se suele utilizar para realizar las fotografías estereoscópicas. Cuando la distancia entre las fotografías tomadas para formar la imagen estereoscópica es bastante elevada hablamos de **Hiperestereoscopia**. Es lo que ocurre en el caso de las fotografías aéreas donde se suelen tomar distancias que permitan dar la sensación de ver el objeto como si se estuviera a tres metros de distancia del mismo.

Existe una fórmula con la que podemos calcular la distancia que debe existir entre las diferentes tomas, para obtener una imagen estereoscópica donde se pueda apreciar la sensación de relieve de los objetos a gran calidad.

(0,065m/3m) / (d/L)

- 0,065m distancia interocular.
- 3m distancia a la que se quieren observar los objetos en la imagen virtual.
- d distancia que queremos calcular.
- L distancia entre el punto desde donde se realizan las fotografías y los objetos que queremos captar en la imagen.

Despejando d, tenemos que d= 0,021667 x L, aproximadamente d = (1/50) x L, por tanto si el avión estuviera tomando imágenes a una distancia de 6 km el intervalo de tiempo que se debería producir entre captura y captura debería ser de 120 metros.

Como vemos es muy sencillo calcular la distancia que hay que recorrer para tomar una nueva fotografía, no sería complicado por tanto poder calcular las distancias que existe entre elementos que aparecen en la fotografía. Esta técnica es utilizada también en la fotogrametría donde se realiza la medición de distancias a partir de las fotografías tomadas.

Hemos mencionado la **Hiperestereoscopia**, el fenómeno contrario es conocido como **Hipoestereoscopia** que sucede cuando existe poca distancia entre el punto donde se realizan las fotografías y los objetos a fotografiar pudiendo provocar que aquello que queremos fotografía salga fuera del fotograma. Para calcular la distancia adecuada nos vale la misma formula, simplemente vasta con variar la distancia a la que queremos ver los objetos en la imagen virtual, por ejemplo:

(0,065m/0,25m) = d/L despejando **d** tenemos d = 0,25 x L

Principios matemáticos, calculo de distancia focal

Si queremos crear una imagen estereoscópica con dos cámara fotográficas, lo primero que tenemos que calcular es la distancia a la que deben estar las lentes de las dos cámaras, es lo que conocemos como distancia focal o stereo base.

Para realizar el cálculo necesitamos tener en cuenta los siguientes aspectos:

- La distancia entre los objetos dentro del campo de visión debe ser la adecuada, es lo que conocemos como profundidad de campo o depth of field.

- Asegurarse de que el "Stereo parallax" no excede 1.2mm (formato de 35mm) o 2.3mm (en el formato de 60mm). Stereo Parallax es la distancia que existe entre los objetos cercanos y los lejanos en la vista utilizada. Stereo Parallax suele ser expresado como angular parallax o linear parallax.

- Angular parallax se refiere al grado de convergencia necesario para fundir un objeto. Un objeto en el infinito estereoscópico tiene una angular parallax de cero en el mundo real.

- Linear parallax es la diferencia que existe entre la distancia de los puntos homólogos en el par estéreo. Recordemos que cuando tomábamos dos fotografías para construir la imagen estereoscópica existían gran cantidad de puntos en común en ambas y puntos únicos en cada una de ellas.

- Seguir los principios estéticos de una buena imagen estéreo.

Tanto la profundidad de campo como el stereo parallax son conceptos matemáticos que pueden ser calculados fácilmente, mientras que los principios estéticos que se sigan dependerán únicamente de la creatividad y capacidad artística de cada persona.

Para realizar el cálculo de la distancia focal utilizamos la fórmula desarrollada por Jhon Bercovitz obteniendo el máximo efecto estereoscópico en las imágenes dentro de lo permitido.

$$B = P/(L-N) \left(LN/F - (L+N)/2 \right)$$

- B: Base estéreo (distancia entre los ejes ópticos de la cámara)
- P: "Parallax" obtenido, en milímetros
- L: Distancia mas lejana de las lentes de la cámara
- N: Distancia mas cercana a las lentes de la cámara
- F: Distancia focal de las lentes

La profundidad del sujeto suele coincidir con $L - N = D$ aunque no siempre se cumple, sobre todo con distancias grandes. Según Bercovitz , cuando la distancia al objeto más alejado de la cámara sea infinito podemos utilizar: $B = P\ (N/F - ½)$, pudiendo aplicarla para vistas ortogonales.

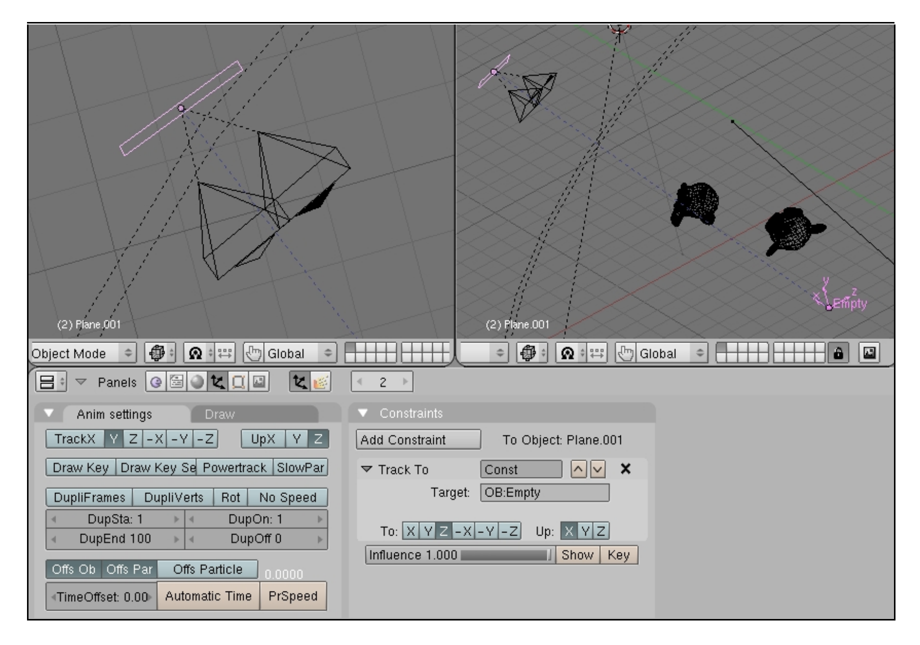


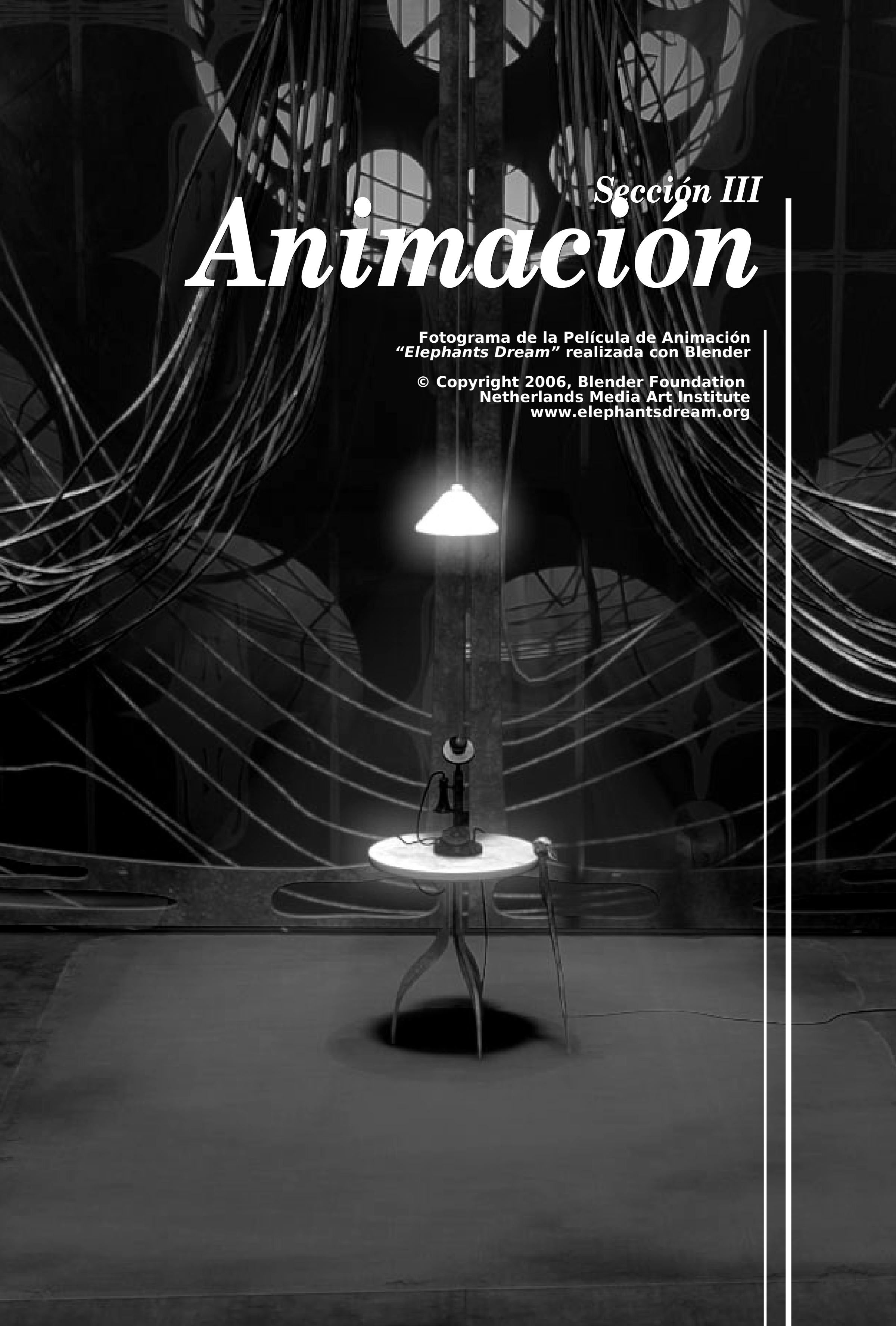
Figura 3. *Configuración para render Estereo en Blender.*

Configuración en Blender

La configuración en Blender es bastante sencilla. Basta con añadir dos cámaras a la escena que estén emparentadas con un objeto que controlará su rotación y posición. Para ello, este objeto padre tendrá una restricción de tipo "*Track To*" con un objeto Empty (ver Figura 3). El render habrá que realizarlo con cada una de las cámaras, componiendo el resultado de las dos cámaras en una misma imagen o fotograma del video resultado.

Referencias

- `http://dmi.uib.es/~abasolo/cursorealidad/paco/Estereoscopia.html`
- `http://www.neoyet.com/vision.htm`
- `http://nzphoto.tripod.com/stereo/3dtake/fbercowitz.htm`
- `http://www.users.red3i.es/~stereoweb/vision.htm`



Sección III
Animación

Fotograma de la Película de Animación
"Elephants Dream" realizada con Blender


IPO
A monkey curve

Técnicas de Animación en Blender

Carlos López Yrigaray
klopes@unizar.es ::



El arte ha encontrado en la infografía su última forma de expresión, y la generalización de esta ha llevado a la aparición de un sinnúmero de artistas que, asu vez, han sabido transmitir su obra a través de la red. Recíprocamente, la popularidad de los programas gráficos, y en particular los que favorecen la creación de personajes y su animación, han impulsado la creación de programas de licencia libre como Blender y la mejora constante del software disponible en todos los ámbitos.

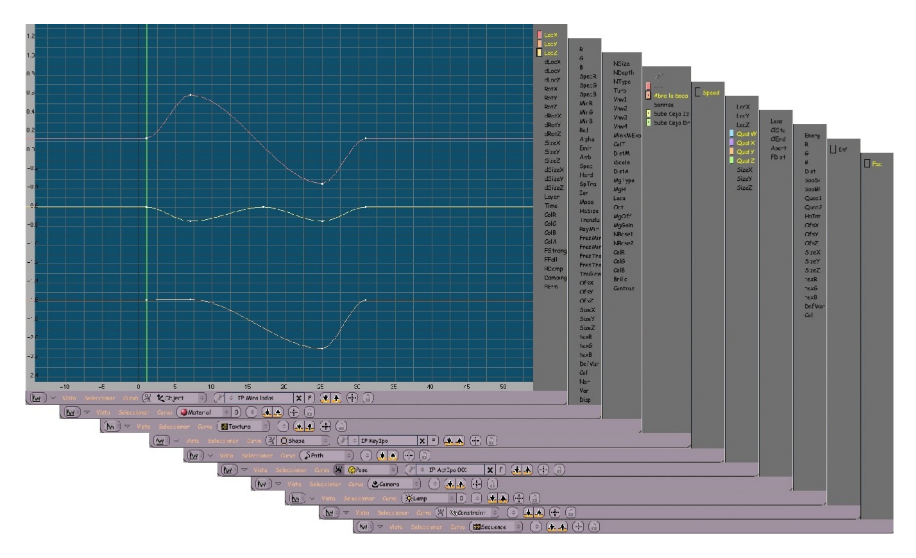
Estas mejoras se hacen especialmente patentes en cuanto a la facilidad de uso de las utilidades, permitiendo la creación de escenas expresivas con un esfuerzo menor, sin menoscabo de la capacidad artística, que se ve por el contrario más libre en cuanto a las limitaciones del programa.

Como el proceso de modelado, que ha pasado a dividirse en varios tipos (desde la creación de estructuras malladas polígono a polígono hasta la escultura virtual usando herramientas adecuadas), o la creación de materiales y entonación de la luz (que van pasando por sucesivas fases de calidad, refinándose para acercarse cada vez más a la realidad de una fotografía, o haciéndose tan manejables como para modelar el ambiente de forma creativa), el de la animación ha evolucionado hacia el modulado de la composición de acciones en una línea de tiempo, o la implementación de sistemas de toma de decisiones de los propios personajes mediante algoritmos de inteligencia artificial.

En esta exposición se pretende comentar el estado del arte en el momento de escribir estas líneas, en cuanto a la animación en Blender 3D, desde el movimiento de un objeto hasta las posibilidades de animar un personaje.

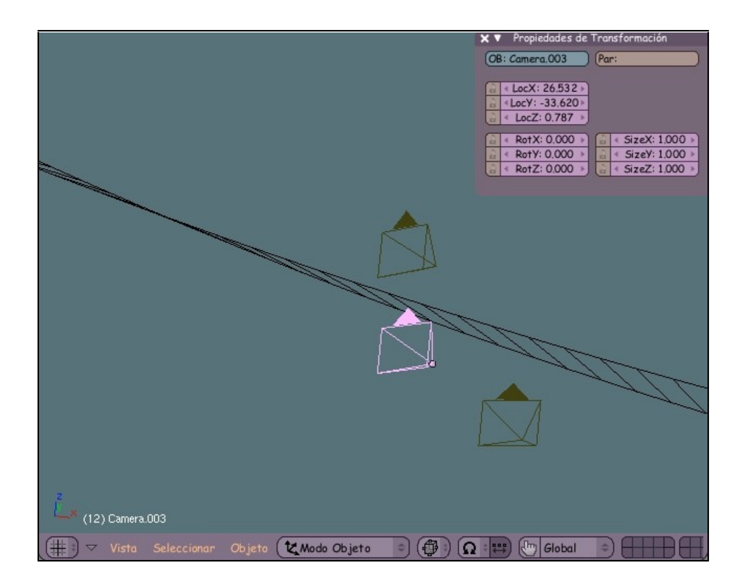
Moviendo Blender

El movimiento de un objeto lo definiremos tal como se usa en su sentido matemático, es decir: su desplazamiento por el espacio 3D, junto con sus giros y sus cambios de escala en los tres ejes de referencia, esto es: la modificación de su matriz característica.



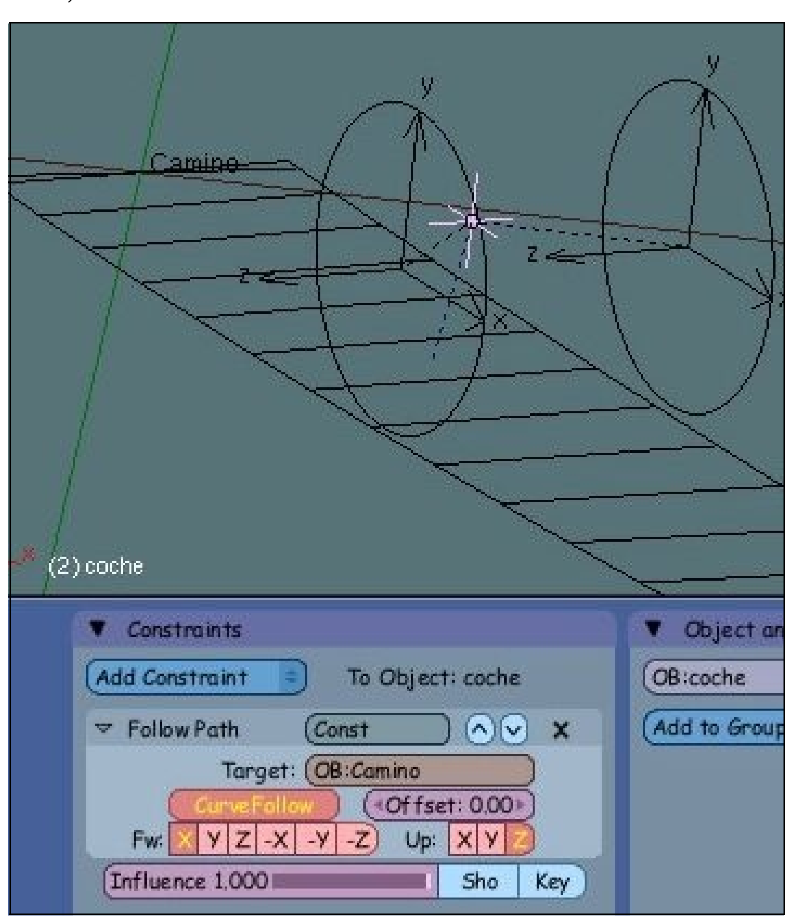
A cualquier otro cambio aparente del objeto (cambios en su geometría, deformación esquelética o por curvatura del espacio, roturas, color, material...) lo llamaremos, en general, animación, y ya está.

Inicialmente, un movimiento se describe mediante curvas que indican el valor del parámetro en el tiempo, ya sean valores coordenados, ángulos, intensidades... Si un objeto sólo va a dar botes por la escena sin moverse del sitio, sólo habrá que definir una curva en el canal LocZ (figura superior).



Estas curvas se agrupan en objetos de la llamada clase IPO, y sus curvas son las curvas IPO. Podemos ver copias del objeto animado en cada uno de sus puntos de cambio (nodos, keys) para hacernos una idea de su evolución.

Los objetos también pueden estar animados simplemente por restricciones, cambiando su posición, orientación y aspecto por la acción de otros elementos, como el conservar la orientación hacia un objetivo, clonar sus coordenadas u orientación, recorrer una curva, ... Estas restricciones también están controladas a su vez por una curva IPO que las puede desactivar cuando su valor es 0. También podemos generar animación con modificadores: enganchando vértices a otro objeto (Hooks), pasando a través de una rejilla deformada (lattice), construyendo un esqueleto,...



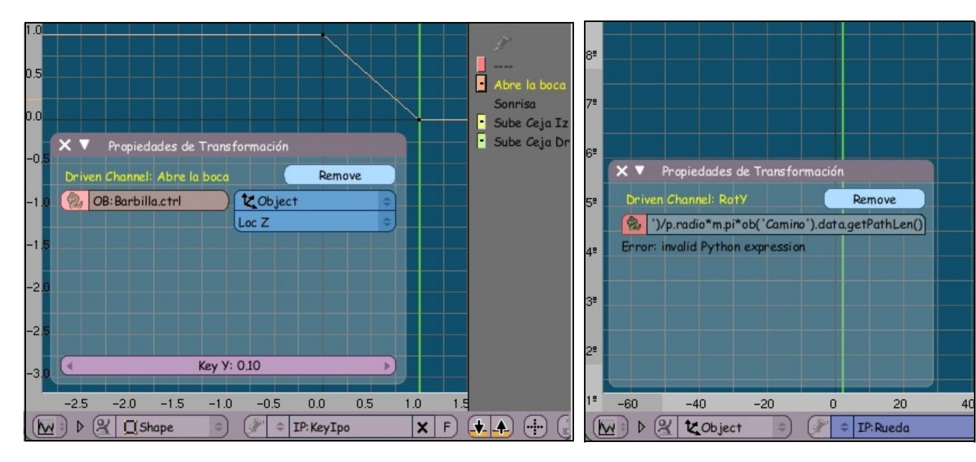
Python

Evidentemente, existiendo un entorno de programación dentro del propio Blender (un API para lenguaje Python), estamos capacitados para crear curvas IPO mediante un programa, explícitamente o en base a un interface controlado por el usuario. El movimiento, entonces, estaría calculado previamente en el momento de uso del script. Pero también podemos evitar el paso de crear curvas modificando directamente las características del objeto en cada fotograma. Para ello hacemos uso de los ScriptLinks, enlaces directos desde una entidad de blender (un objeto3D, cámara, material... o una escena completa) a programas que se ejecutarán en cada cambio de cuadro.

Dirigiendo el movimiento externamente: Drivers

Otra de las últimas herramientas añadidas en Blender son los drivers, que controlan curvas IPO individualmente por medio de otra curva, o bien con una línea o un programa escrito en Python (PyDrivers). A grandes rasgos, nos permite crear animaciones que dependen de otras animaciones o de movimientos de objetos.

Como era de esperar, todas las curvas IPO de todo tipo pueden ser controladas por drivers, también las que cambian los parámetros de los materiales, luces y cámaras, y las que controlan los gestos en las RVKs (ver más abajo).



Aprovechando el Game-Engine

Otra de las utilidades recientemente implementadas es la generación de movimientos generados por el motor de juegos de Blender, lo que nos da la posibilidad de añadir complejidad física a la escena con colisiones y comportamiento realista de los objetos.





Esqueletos en Blender

El más potente modificador, casi imprescindible, para la creación de personajes animados, es el uso de esqueletos, en Blender llamados Armatures, que constan de segmentos (huesos, Bones) que pueden emparentarse formando una estructura móvil que se puede animar.

A la preparación de un esqueleto completo, adecuado al objeto que pretende deformar, con sus controles para usar de manera cómoda, se le llama rigging.

Acciones

El movimiento de los huesos responde a IPOs de un tipo especial, Actions, que permite desplazamiento, rotación y escalado de estos individualmente. Hay que hacer notar que las rotaciones utilizan 4 parámetros, según el método de orientación de los cuaternios de Hamilton (W, X, Y, Z) en lugar de los ángulos de Euler (X, Y, Z) que usan los objetos 3D, para aumentar la velocidad de cálculo y evitar los ejes privilegiados y puntos nodales.



La situación de los huesos de un esqueleto en un instante forma una Pose, y una secuencia de poses genera una Acción. Así que podemos crear una secuencia completa animando un esqueleto en una acción única con poses sucesivas, si bien podemos pensar que ciertas partes de la acción pueden repetirse, en esta u otras acciones (menear la cabeza, aplaudir, dar un paso), con lo que lo ideal es descomponer la animación en acciones sucesivas, que pueden mezclarse unas con otras, más cuando en una acción no tienen por qué intervenir todos los huesos, así que podemos crear acciones para los pies, acciones para los brazos, para los dedos, la cabeza, la cola...

En las últimas versiones de Blender no sólo podemos incluir secuencias de poses de esqueletos, sino que podemos combinarlas con movimientos de objetos o de la propia Armature como objeto en sí mismo. Esto adquiere sentido si tenemos objetos conectados al esqueleto o al modelo que los están controlando, como complementos de la ropa que se mueven al caminar, objetos de control que tiran de los ojos hacia adelante en una acción de sorpresa, etc.

Deformación de mallas

Esqueletos

A estos esqueletos podemos asociar una malla para que se deforme con el movimiento de los huesos, aunque también podemos usarlos emparentando objetos directamente a los huesos, como componentes

rígidos robóticos. La deformación de una malla asociada a un esqueleto (llamada skinning) se puede producir de varias maneras, ya sea mediante grupos de vértices que siguen a determinados huesos, por asignación automática de cercanía a estos (Envolventes), o una combinación de ambas técnicas.

Vertex Keys

Esta herramienta es la más importante para dar expresividad a nuestros personajes, pues es la manera más cómoda de modificar las mallas en una secuencia para que cambien su apariencia con el tiempo. Consiste en crear copias modificadas de la malla original (gestos, Shapes), de manera que podemos conseguir una colección de gestos para mezclarlos y combinarlos en la animación. Si bien existe una forma básica de usar las Vertex Keys, de forma absoluta, haciendo uso de ellas una tras otra linealmente, la opción más interesante es usarlas de forma relativa (RVKs). Esto significa que en cada momento, estamos sumando los gestos de una cara que nos interesan, para conseguir que un personaje pueda hablar, parpadar y mover las cejas a la vez. El mecanismo es simple de entender, y ahora muy fácil de gestionar gracias a los botones de la ventana de acciones: cada gesto está controlado por una curva IPO, que indica con cuánta intensidad se realiza: con 0 no tiene lugar, con 1 vemos el gesto tal cual, y con valores superiores podemos incluso exagerarlo.

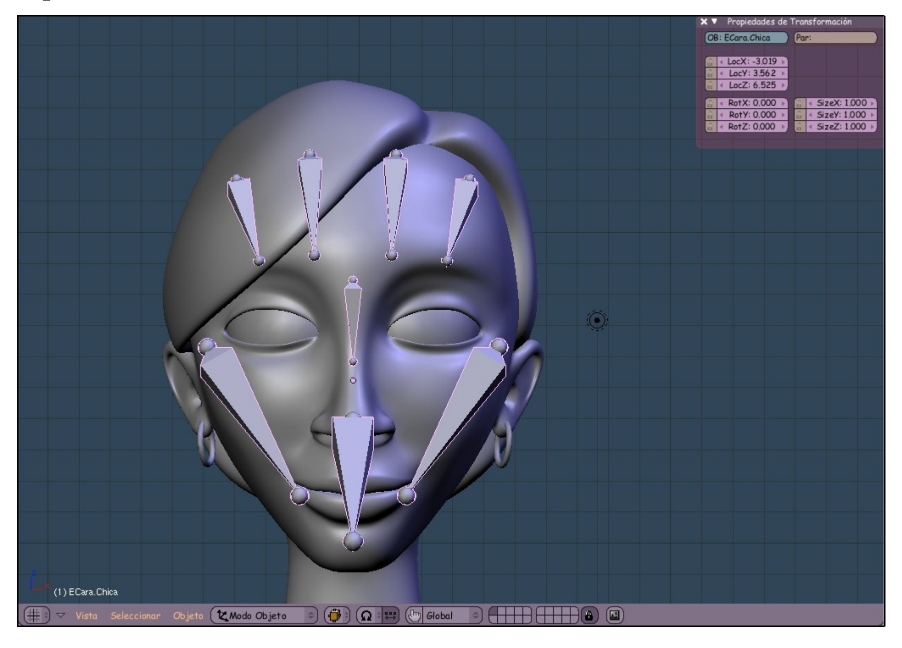
Gracias a la posibilidad ya comentada de introducir animaciones de objetos dentro de acciones, también podemos añadir secuencias de RVKs dentro de estas, consiguiendo sumar la expresividad de la malla a los movimientos de los huesos.

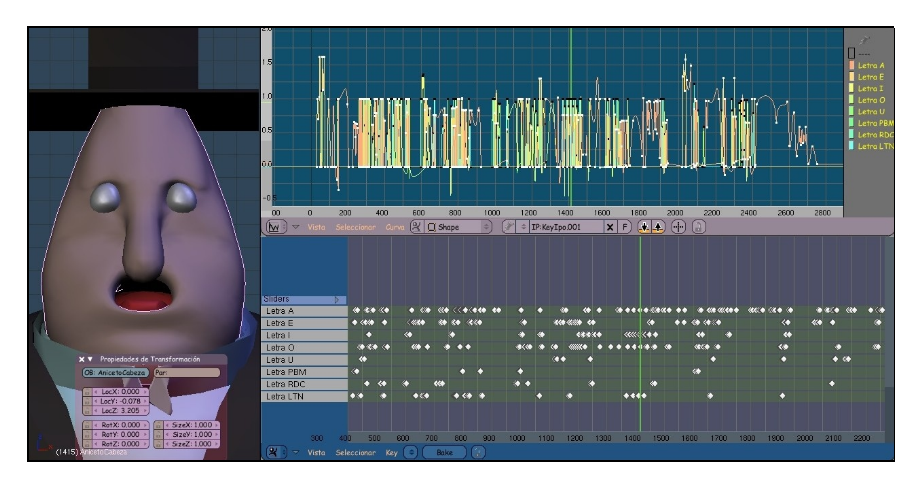


Animación facial y sincronización labial

Como apartado especial quedan estas actividades, imprescindibles para dar expresividad y lenguaje a nuestros personajes y que por sí mismas constituyen una ocupación en la organización de un proyecto animado. La expresividad es responsabilidad del animador y de su capacidad artística, y la sincronización labial (lipsync) exige un esfuerzo a veces impredecible, y conocimientos del habla y la fonación en el idioma en el que habla el personaje. Supondría material para una charla que, por otra parte ya se impartió en otro encuentro, así que en este apartado nos limitaremos a comentar lo que ofrece Blender para cambiar la expresión de la cara de un personaje. En general, podemos usar las dos técnicas comentadas de forma no excluyente.

Un rigging facial es adecuado para los motores de tiempo real (MTR), esto es: juegos y aplicaciones interactivas. En la actualidad, raramente un MTR permite combinaciones de mallas para crear expresividad o habla, por lo que todas las deformaciones deben hacerse con esqueletos. Esta tarea no suele ser complicada porque normalmente no se exige una interpretación dramática del actor, pero si fuera necesario, el método consiste en crear un sistema de huesos análogo a la musculatura de la cara, de manera que al asociar los vértices, estos se muevan como la piel que hay sobre ellos. La aproximación es bastante buena, pero no es un método adecuado para expresividad sutil.



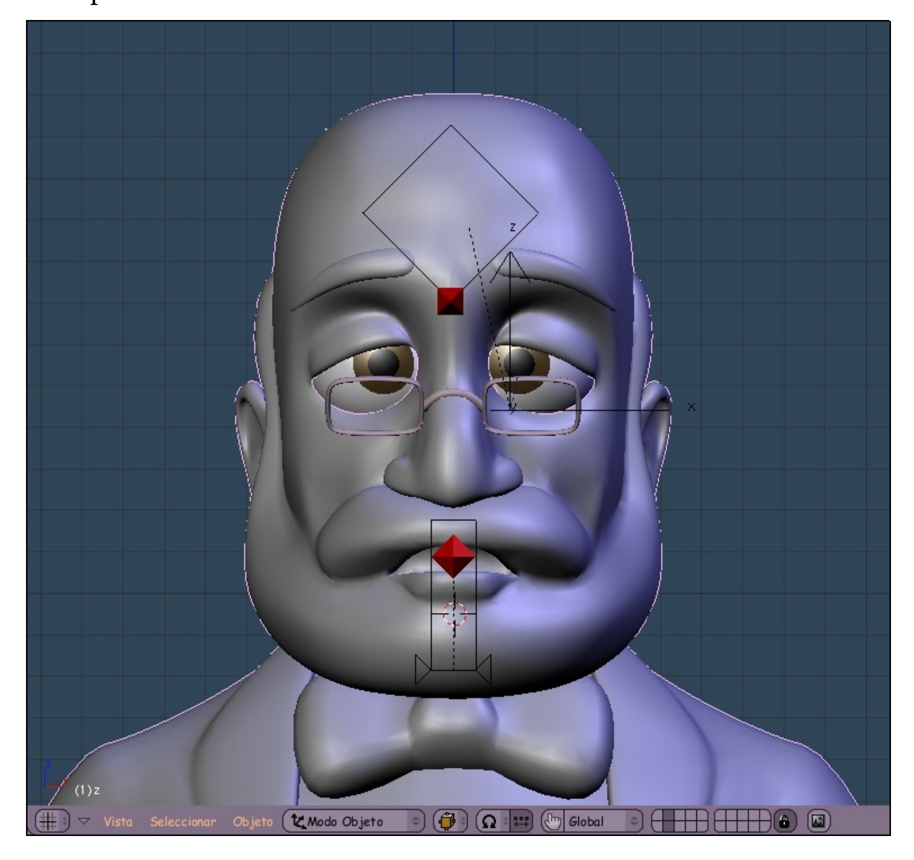


La manera habitual de dar expresiones a una cara es modelarlas mediante copias de la malla, separadas en mitades si es necesario (ceja derecha arriba - ceja izquierda arriba,...). Estas mallas, en realidad se guardan internamente como deplazamientos relativos de los vértices con respecto a la malla base, de manera que el programa sólo tiene que sumar, por cada vértice, un vector por cada expresión activa; este vector estará previamente multiplicado por la intensidad con que se realiza esa expresión, usualmente un número entre 0 y 1. Esta intensidad es, precisamente, la que nos dan las curvas IPO que se definen en las RVKs.

La comodidad para insertar nuevos gestos en la animación es muy importante para un trabajo fluido. Esencialmente, toda la potencia para generar animación por RVKs estaba implementada desde hace años en Blender, pero gracias a los nuevos paneles de Shapes, su inclusión con deslizadores en la ventana de acciones, la asignación de nombres a cada expresión y otros detalles que poco a poco se han ido añadiendo, es cada vez más rápido conseguir lo mismo y con más control.

Creación de GUIs personalizados con drivers

La implementación de drivers que controlen el movimiento ha abierto un sinfín de posibilidades para la animación, y el más inmediato parece ser el control manual mediante creación de botones (sliders) en el propio espacio 3D. Así, el desplazamiento de un objeto puede generar animación en todos los objetos que tengan un IPO con drivers asociados a él. Johnny Matthews (guitargeek), un experto usuario de Blender, creó un pequeño script que inmediatamente fue añadido a la versión oficial. Genera controles de diversas formas que se usan para enlazar a drivers de una malla con RVKs, de manera que podemos fabricar una cara esquemática con pequeños objetos que, al moverlos, nos mueven pequeños rasgos directamente sin tocar para nada el sistema clásico de animación.



Blender y Doom 3

Carlos Folch Hidalgo
FOLKYTO@teleline.es ::



Hoy día todo el mundo ha oido hablar de los videojuegos que realizan empresas famosas como Id Software, Valve, Blizzard y hasta Microsoft. Nombres como Doom 3, Half Life 2, War of Warcraft, etc.. son famosos entre el público general. Pero estos juegos no sólo permiten divertirse de la manera tradicional, contra la "máquina" y online, sino que además abrieron hace tiempo un nuevo campo para el desarrollo Indie: los mods.

Hay que decir que los mods nacieron con juegos mucho más antiguos, pero nos centraremos en los que utilizan engines 3d y requieren de un software 3d para la creación de sus gráficos.

Videojuegos y mods

Un mod no es sino una variación de un juego comercial. La variación puede ser parcial o total. Los mods con variaciones parciales son los más comunes, aprovechándose la mecánica del juego original e introduciendo nuevos gráficos, sonidos, comportamientos y niveles.

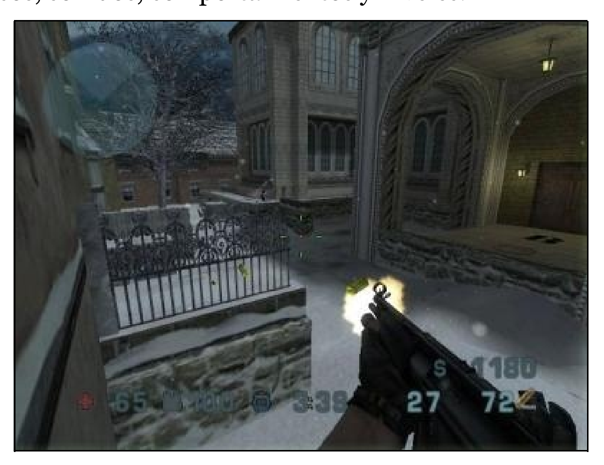


Figura 1. *Captura de pantalla de Counter Strike.*



Figura 2. *Captura de pantalla de Doom.*

Los mods con variaciones totales son más escasos, sobretodo debido a la dificultad y al tiempo de desarrollo necesario. Uno de los mods más famosos de la historia es el *Counter Strike*, que fué desarrollado como mod del *Half Life*.

¿Cómo puedo crear un mod?

El potencial creativo y de diversión extra que proporciona la creación de mods es, hoy día, indudable. Las empresas desarrolladoras de videojuegos hace tiempo que se dieron cuenta de su utilidad y por ello muchas proporcionan algunas de las herramientas necesarias para el desarrollo de mods. Herramientas de compilación y los montadores de niveles son ejemplos de ello. Veamos un par de ejemplos, buscando engines recientes, para no extendernos eternamente:

Id Software

Creadora de Doom y Quake (como títulos más famosos), es una de las empresas más punteras a la hora de favorecer la creación de mods y de apoyar el desarrollo Indie en general. No sólo proporciona editores de niveles con sus juegos, sino que además acostumbra a liberar sus motores y sus herramientas como GPL cuando ha pasado un tiempo desde su publicación.

Como ejemplo saber que id Software liberó el código fuente de Quake III Arena el verano pasado, y que en Febrero de este año liberó el GtkRadiant y el q3map2 (editor de niveles y compilador). Además es una de las pocas

compañías que sacan versiones para Linux de su juegos y sus herramientas, y es conocida su predilección por el desarrollo para Opengl antes que para Directx (ambas son API´s 3D, si bien Opengl es multiplataforma).

No podemos pasar página sin mencionar a John Carmack, co-fundador y programador de id Software, y uno de los programadores que más avances ha introducido en los videojuegos a partir de ideas propias y ajenas.

Valve

Creadora de Half Life 1 y 2, Day Of Defeat, etc.., y uno de los mayores contrincantes de id Software con permiso de Epic Games (Unreal) y de Crytek Studios (Farcry). Valve no tiene una política de contenidos tan abierta como id Software, pero su apoyo a la creación de mods también es bastante bueno. A partir de la salida de "*Counter Strike: condition cero*" incluyeron en sus juegos un sistema multijugador basado en un entorno propio (Steam) que monitoriza sus juegos existentes en la máquina del jugador.

Desde ese entorno se puede lanzar el editor de niveles, que también se puede conseguir de forma independiente. Es el "famoso" Valve Hammer Editor. Además, para la realización de personajes, Valve logró la colaboración de Softimage para distribuir de forma gratuita (freeware) una versión, ligeramente limitada, del conocido XSI, al que bautizaron como XSI Mod Tool (antes Softimage EXP).

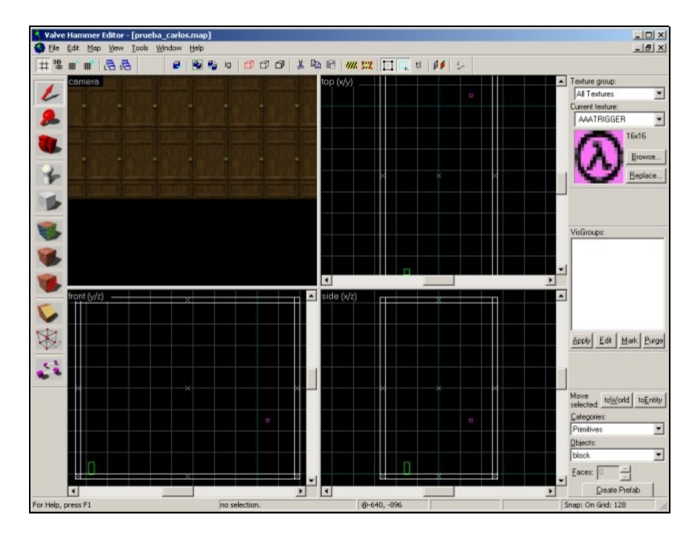


Figura 3. *Valve Hammer Editor.*

El apartado gráfico y Blender

Como ya hemos dicho los mods pueden ser modificaciones que alcancen varios apartados del juego original (gráficos, música, comportamientos, objetivos, etc..), y de todos ellos vamos a concentrarnos en el apartado gráfico, y en la creación de contenidos 3d.

Los editores de niveles, proporcionados por las desarrolladoras, se encargan de mezclar todos los elementos que el “modder” le proporcione. Una vez diseñado un mapa, o nivel, el editor suele tener la posiblidad de compilar el mapa para ser cargado directamente en el engine original del juego (llamado a través de cónsola, o con una GUI especial para ello). Es la forma de mod más sencilla, la creación de mapas y su carga directa en el juego original (tambien llamados **packs**)

Cada editor de niveles suele tener su propio código y funcionamiento. Las “entidades”, los “props”, los “info_player_start”, los “triggers”, etc.. son sólo un ejemplo de términos que podemos encontrar en distintos editores. No vamos a entrar en ellos, porque tendríamos que hacer un glossario para cada editor (y para eso ya están las ayudas y los tutoriales) y vamos a simplificar en la creación de 2 componentes que siempre encontramos, denominados de distintas maneras, en los editores de mapas:

- Objetos (estáticos o animados)
- Personajes animados

Los objetos cumplen varias funciones. Desde formar parte del suelo, paredes y techo (o cielo) del mapa hasta ser simples objetos de ambientación. Si van a tener propiedades de sombreado, afectación de físicas o si van a ser fraccionables o con decals (texturas que se añaden cuando les afecta una acción) no lo vamos a tratar aquí.

Los personajes animados (colocar un personaje no animado es colocar una estatua, con lo que ya no es un personaje :P) también pueden cumplir varias funciones, pueden verse afectados por comportamientos, seguir rutas, ser activados por otros elementos, etc.. Tampoco vamos a tratar eso. Lo que sí vamos a ver son los elementos, que forman parte de los objetos y de los personajes, que podemos crear o definir con Blender:

- **Modelado Lowpoly**: Creación tridimensional del modelo, en triángulos.
- **Texturas**: Aplicación de mapas UV de texturas en canal diffuse y normal maps.
- **Esqueletos**: Creación de Armatures para animar los modelos.

Creación de un personaje con Blender

Para la creación de nuestro personaje utilizaremos referencias y bocetos. El bocetado de un sketch, o idea rápida, nos dará seguridad y rapidez a la hora de modelar.

Modelado del personaje

Mientras modelamos podemos modificar aspectos que creamos mejorables. El boceto no sólo suele ser una buena forma de trabajo propia, también es la forma de enlazar el trabajo del artdesigner con el del modelador. A los bocetos que se crean de frente y de perfil, optimizados para su uso directo en el modelador usado, se les conoce comunmente como blueprints.

En Blender podemos cargar dichos blueprints directamente en los visores. A través del menú **View** podremos colocar imágenes distintas en cada división, de la ventana 3d, que hagamos.

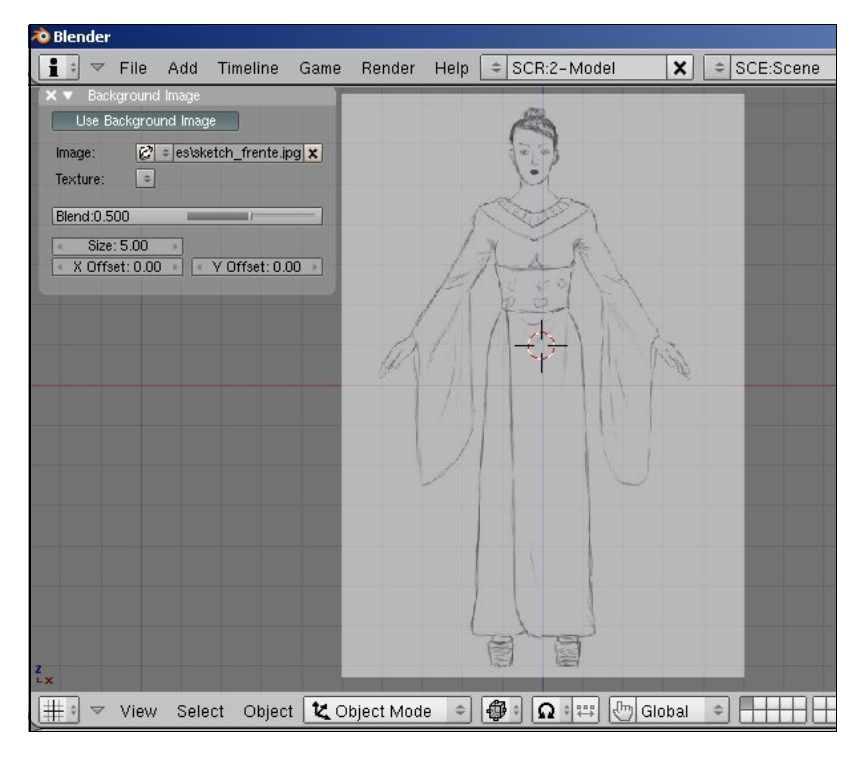


__Figura 4.__ Boceto cargado en la 3D View de Blender.

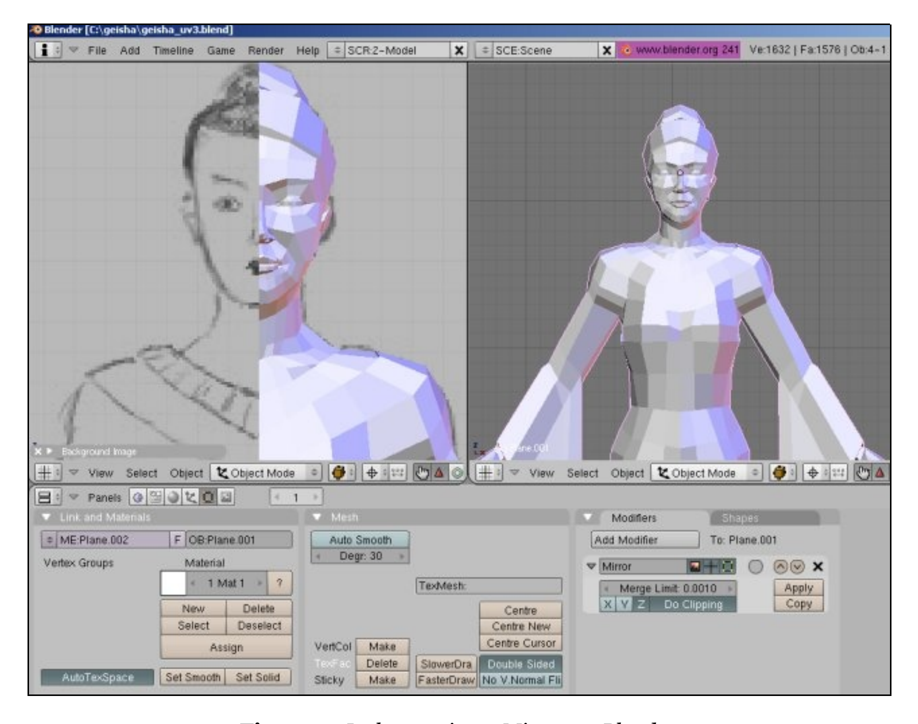


Figura 5. *La herramienta Mirror en Blender.*

Para modelar el modelo podemos usar la técnica que deseemos, mediante Box modeling, usando extrusiones para ir creando el personaje o incluso vértice a vértice (o el concepto "brick by brick" de Jbelanche). Independientemente de la técnica usada hay una herramienta que resulta muy útil, en cualquier software 3d, a la que se le llama **Mirror**.

El **Mirror** no es sino el reflejo, con respecto a un eje, de lo que nosotros vamos modelando, y se suele usar modelando un lado del personaje y dejando que el Mirror crée la otra mitad como copia reflejada. Esto permite personajes simétricos y sólo lo desactivaremos a la hora de crear irregularidades (como en el peinado), personajes asimétricos o a la hora de añadir determinados tipos de ropa y complementos.

Por último hay que señalar una condición especial del modelado de personajes para videojuegos. Tenemos la necesidad de conseguir mallas ligeras para liberar al motor gráfico y que sea así capaz de mostrar en pantalla todo lo necesario sin que se relentice (fps) el juego. A estas "mallas ligeras" se las conoce como modelado Lowpoly, y su base consiste en intentar modelar sólo aquellos detalles que sean imposibles de simular mediante textura o normal map.

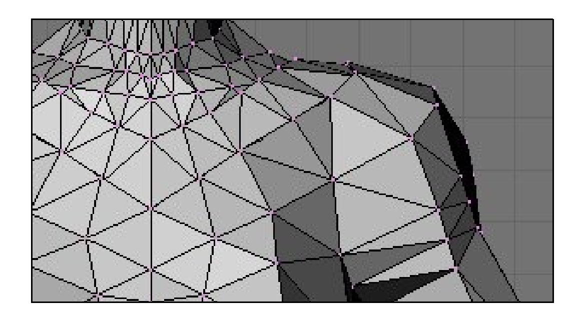


Figura 6. *Malla triangulizada en Blender*.

Actualmente un personaje protagonista puede rondar tranquilamente los 3000 polys (tris) y los personajes secundarios pueden ir de 1000 a 2000 polys (tris). Obviamente si hay personajes que sólo actuarán al fondo de la acción y que nunca se verán en detalle estas cifras pueden bajar mucho, también hay que señalar que los actuales motores de última generación permiten personajes principales de más de 5000 polys.

El término polys, usado aquí, hace referencia al número de polígonos, y el términos "tris" se usa para especificar que dichos polígonos son triángulos. En la mayoría de softwares de creación 3d podemos crear polígonos de 4 o más lados (usando f-gons o n-gons) pero para la exportación para videojuegos debemos dejar la malla triangulizada porque los motores gráficos, y más concretamente las API´s gráficas usadas, sólo muestran triángulos. En Blender hay la opción de triangulizar una malla pulsando *Ctrl+t* pero siempre es mejor hacerlo de forma manual para evitar aristas rebeldes.

Creación del mapa UV

Una vez tenemos el personaje modelado hay que aplicarle texturas. Las texturas usadas dependen, en cuanto a tamaño y tipo, de la capacidad del motor. Las texturas deben tener normalmente un tamaño que sea potencia de 2, aunque dependiendo de la aceleradora pueden usarse texturas que se salten esa norma pero eso no lo vamos a explicar aquí. Así se suelen usar texturas de 16x16, 32x32, 64x64, 128x128, 256x256, etc... El tamaño máximo que vamos a poder usar dependerá de que el motor sea capaz de mostrarla, junto con el resto de mallas y texturas, en una escena sin relentizarse. Hoy día 512x512 ó 1024x1024 son tamaños buenos para hacer la textura de nuestro personaje. Mencionar que aunque en los ejemplos he puesto números simétricos las texturas pueden ser de 32x64, etc...

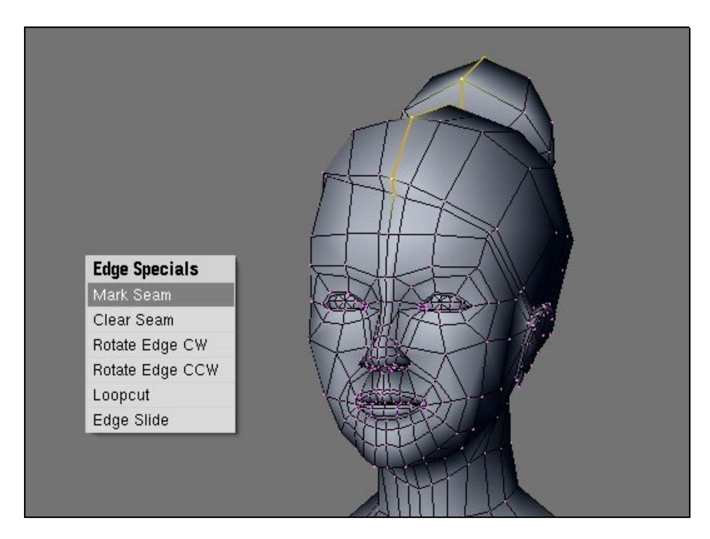


Figura 7. *Usando el Mark Seam en Blender.*

En cuanto al tipo, las texturas más usadas son las que se usarían como diffuse (o el color propiamente dicho) y las que nos darían la información de bump (relieve) y las especulares (brillo). También hay motores que aceptan los normal maps, de los que hablaremos después. A la hora de plantear nuestra textura debemos tener en cuenta si deberá ser, o no, tileable (que se puede repetir, una junto a otra, sin que se vean las uniones).

Para poder aplicar las texturas a nuestro personajes debemos definir antes el mapa UV. Dicho mapa no es sino extensión de las coordenadas 3d, de los vértices de la malla, en un mapa 2d. Para crear dicho mapa en Blender usaremos el modo Face Selected Mode, y eligiremos el tipo de proyección UV. Si usamos el LSCM marcaremos algunos vértices por donde el programa calculará un corte (mark seam) y luego ajustaremos las coordenadas UV, en el Image Editor, mediante el uso de Points y el LSCM Unwrap. Cuando el modelo es complejo, por ejemplo con un personaje para videojuegos :P, es recomendable ir mapeando por partes en vez de intentar hacerlo todo entero de una tirada.

Cuando tenemos el mapa UV creado lo exportaremos a una imagen que nos muestre el mapeado para poder, posteriormente, pintar encima las texturas en nuestro editor de imagen preferido (ya sea Gimp, Artrage, etc...). Para exportarlo usaremos el script, que trae por defecto Blender, llamado "*Save UV Face Layout*". Posteriormente bastará con que carguemos en Blender la imagen creada y ya podremos ver como queda, perfectamente ajustada, a nuestro modelo (Alt+z para visualizar las texturas aplicadas con UV).

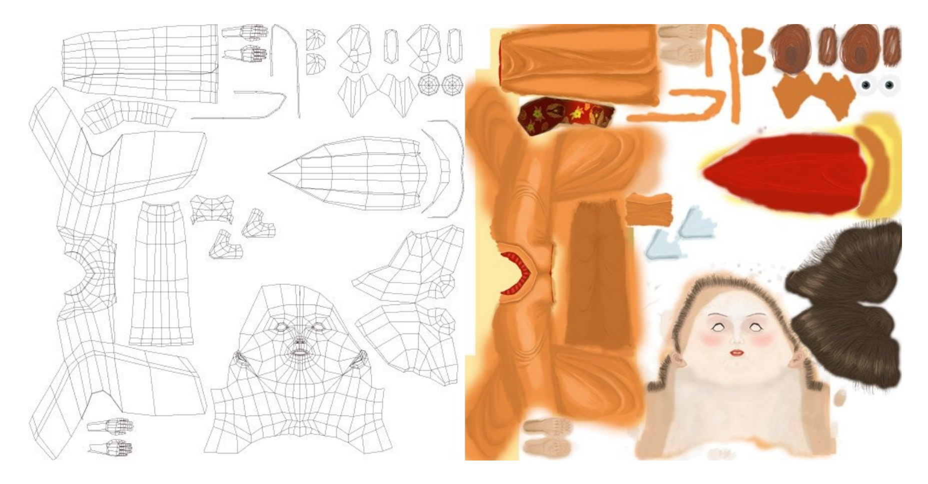


Figura 8. *Mapa UV estraido con el Script "Save UV Face Layout" y pintado con Gimp.*



Figura 9. *El modelo final con la textura aplicada.*

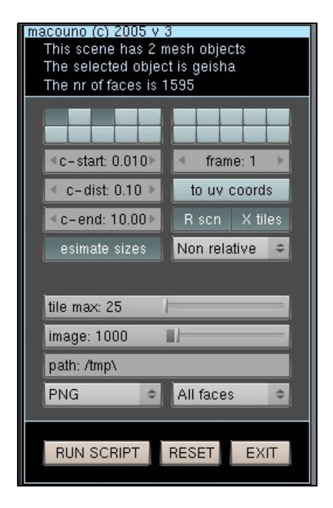


Figura 10. *Script para hacer un "Render to Texture".*

Render to texture y Normal mapping

Ya tenemos el mapa UV y las texturas, pero no nos convence la sensación de volumen que hemos conseguido con ellas. Es hora de usar el "render to texture". Dicho término se usa para referirse al proceso de render, de cada una de las caras del modelo afectadas por la iluminación que deseemos, y al volcado de esas imágenes en el mapa UV de nuestro modelo. De esta forma, y dependiendo del motor de render usado, podemos conseguir un sombreado más realista en nuestras texturas. Hay que advertir que se suele usar para ello una iluminación que simule una GI suave para permitir que nuestro modelo sea válido en cualquier situación. En Blender disponemos de un script que realiza esto, con mayor o menor fortuna; es el BrayBaker[φ], realizado por *macouno*.

No vamos a despedirnos aún de este script, ya que permite también la creación de normal maps para nuestro modelo. Los normal maps son un concepto parecido a los bump maps (mapas de relieve en escala de grises), pero a diferencia de éstos últimos los normal maps utilizan la información RGB, con los valores de sus canales asignados a cada eje de coordenadas, para calcular la simulación de relieve en el espacio tridimensional. Esto, más o menos traducido, quiere decir que la simulación de relieve se verá también afectada por la luz de la escena y por la posición de la cámara con respecto a las caras del objeto (y sus normales ;-))

φ El Script BrayBaker, junto con su documentación puede encontrarse en la siguiente página web: `http://www.alienhelpdesk.com/index.php?id=22`

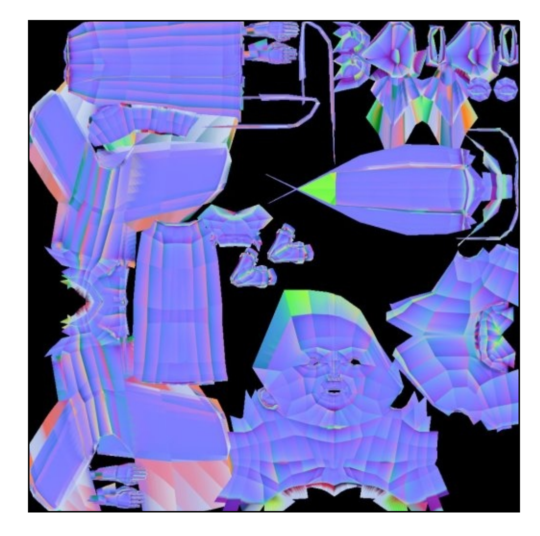



Figura 11. Izqda: *Normal Map extraido con BrayBaker.*
Drcha: *Normal Map aplicado a coordenadas UV.*

La forma de hacerlo es creando un modelo en alta (highpoly) a partir del modelo lowpoly. A este modelo en alta le aplicaremos un material, para crear el normal map, con un color RGB por cada dirección y posteriormente usaremos el script BrayBaker, seleccionando el modelo en lowpoly y manteniendo activo, en el script, el layer donde se encuentra el modelo highpoly. Para la creación del material para normal map lo mejor es revisar la documentación oficial de Blender en la siguiente dirección: `http://blender3d.org/cms/Normal_Maps.491.0.html`

Creación de esqueletos

Con el personaje modelado y texturizado es hora de crear las animaciones que utilizará el motor para que el juego sea mínimamente dinámico (nadie dijo que esto fuera un camino de rosas). Para conseguir eso primero debemos crear una Armature que hará las veces de esqueleto de nuestro modelo. La creación de Armatures puede parecer una incógnita total, pero aveces basta con fijarse en la anatomía real para saber dónde y cómo hay que colocar cada Bone (hueso). Los huesos, que componen las Armatures, son objetos no renderizables unidos entre sí por jerarquías. Se componen de root y tail (cabeza y cola), aunque también se le llama tip a la tail. En el root de cada hueso se encuentra su punto de pivote. Las transformaciones que hagamos, en un nivel de la jerarquía, descenderán hasta sus descendientes transformándolos a todos. Por ejemplo, si rotamos un hueso padre todos sus descendientes rotarán tomando como punto de pivote el root del hueso padre.

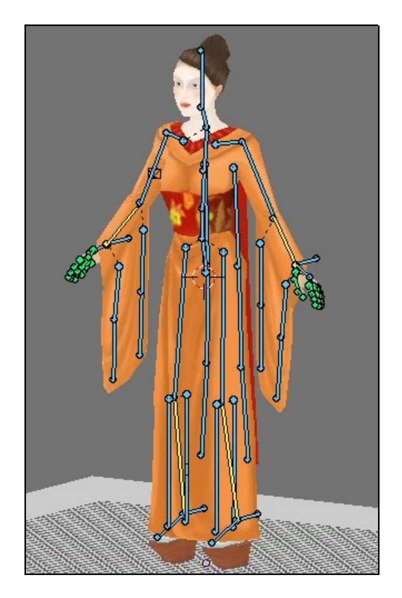




Figura 12. Izqda: *Armatures en Blender*.
Drcha: *Envelopes en Blender*.

Teniendo en cuenta esos aspectos crearemos nuestra Armature cuidando que la jerarquía se comporte también de forma anatómicamente correcta (o mecánicamente correcta si se trata de un robot :P).

Skinning + pesado de vértices

¿Ya puedo empezar a animar mi Armature? No. Aunque ya queda menos. Una vez creada la Armature es hora de enlazar el modelo a la misma. A esto se le llama Skinning y no es sino la asociación de cada vértice a, como mínimo, un hueso. Una vez asociados todos los vértices podremos animar el esqueleto y los vértices se comportarán como si estuviesen pegados a los huesos, osea rotarán, se moverán y se escalarán si los huesos a los que están asignados lo hacen.

Desde la versión 2.40 disponemos en Blender de 4 modos de visualización de la Armature. Uno de ellos, el modo Envelope, nos puede ayudar en el skinning de nuestro modelo. Antes sólo había una forma de hacerlo, que gracias a Dios aún existe, y era mediante la creación manual de Vertex Groups a los que se asigna el mismo nombre que el hueso que les va a influir (cada vértice puede estar en más de un grupo). Si usamos la Armature tipo Envelope no sólo vamos a ver como nuestro esqueleto se parece un poco más al muñeco Michelín, sino que además veremos como la Armature será capaz de influenciar a los vértices que queden dentro de su

estructura o de su zona de influencia. La estructura de los envelopes es bien visible, son esas esferas hiperdesarrollada, su manipulación es individual para el root y para la tail y su uso se basa en el escalado de sus partes hasta que los vértices que nos interesan quedan dentro de la estructura. Además tenemos la zona de influencia soft, cuyo porcentaje de influencia disminuye, de forma cuadrática, conforme nos alejamos del hueso en sí.

Hemos visto que los vértices no sólo pueden estar asociados a un único hueso, pueden ser influenciados por más de un hueso a la vez y aquí es donde empezamos a encontrarnos con el concepto de "pesado de vértices". Si un vértice está asociado a dos huesos podemos decir que recibe una influencia del 50% de cada hueso. Esto se puede manipular mediante el pintado de pesos. Se puede pintar directamente sobre la malla la influencia que queremos que cada hueso haga sobre los vértices, variando así su comportamiento a la hora de las transformaciones. Esto no se hace porque sí, su objetivo es conseguir evitar deformaciones no deseadas, como brazos que al doblarse parece que se rompen en vez de estar flexionándose. Podeis encontrar información sobre skinning y pesado de vértices en la documentación oficial de Blender[10] o en la Mediawiki.

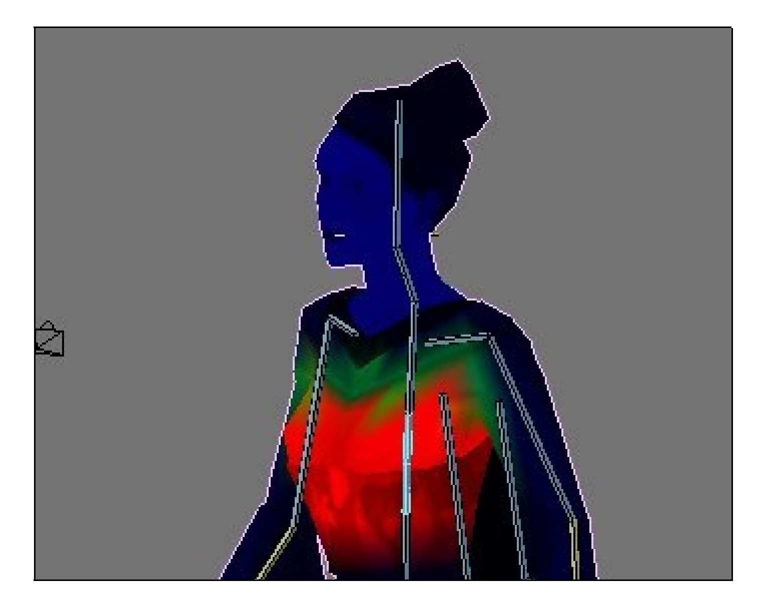
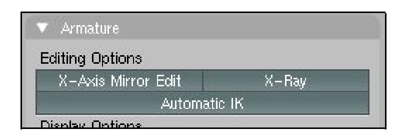


Figura 13. *Pintando pesos en Blender.*

10 http://blender3d.org/cms/Armature_Envelopes.647.0.html
http://mediawiki.blender.org/index.php/Manual/PartIX/Skinning



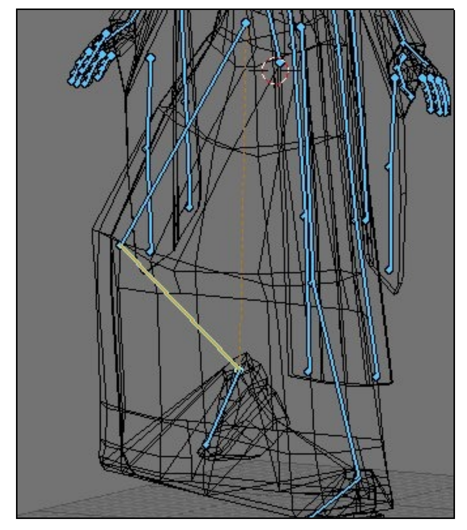


Figura 14.

Arriba: *Automatic IK en Blender.*

Derecha: *Cálculo de efector y root con Automatic IK.*

Cinemática directa e inversa

Vamos a hacer un paso fulgurante por estos conceptos. Normalmente usaremos cinemática directa para nuestros modelos de videojuegos, aunque hay algunos exportadores (y motores) que soportan la cinemática inversa, y no es cuestión de desaprovecharlos.

La Cinemática directa (FK o Forward Kinematics) se basa en la jerarquía directa para sus transformaciones. Éstas bajan por la cadena directamente y se trasladan a todos los descendientes del hueso al que se le aplica la modificación.

La Cinemática Inversa (IK o Inverse Kinematics) utiliza efectores de IK para mover y rotar a la vez toda una cadena jerárquica. Para ello calcula la distancia con el origen de la cadena y rota automáticamente los huesos intermedios cuando se modifica la distancia a dicho origen.

Para aquellos exportadores, y editores, que no calculan las rotaciones usando las cadenas, con cinemática inversa definidas, siempre nos queda la opción de simularla para así aumentar la rápidez con la que creamos las acciones. Para ello Blender tiene la posiblidad de usar el **Automatic IK** que simula la existencia de un efector de cinemática inversa colocado en la punta (tail) del hueso y que toma como origen de la cadena el hueso conectado más alto en la jerarquía.

Hay que saber que la configuración, o Setup, de estos sistemas de animación son un arte en sí mismos y forman parte de una disciplina que se simplifica bajo el concepto de Rigging. Ésta incluye no sólo la creación de la Armature, sino también el pesado de vértices, el uso de constraints, la configuración de FK/IK, el uso de scripts para la creación de GUI´s y ayudantes personalizados, morphing y un sinfín de posiblidades que permiten facilitar el trabajo de los animadores.

Las Actions

Tenemos nuestro modelo texturado y "riggeado", con la armature lista para ser animada. Ahora necesitamos crear nuestra primera Action. Una Action, en Blender, es la unidad donde se guardan las claves de animación de una acción concreta. Para conseguir la sensación de movimiento necesitamos insertar poses de nuestra armature (mediante keys) que varíen a lo largo del tiempo. Blender calculará los intermedios ("inbetwens" en animación tradicional) y suavizará esas transiciones entre pose y pose. Para insertar nuestras poses seleccionaremos la Armature y entraremos en "Pose Mode", una vez allí modificaremos nuestra armature e introduciremos una key para cada hueso pulsando la tecla "i" y seleccionando el tipo de key que necesitemos. Si avanzamos unos cuantos frames, modificamos la pose y volvemos a insertar keys, para los huesos necesarios, observaremos como se produce la animación al desplazar la línea de tiempo.

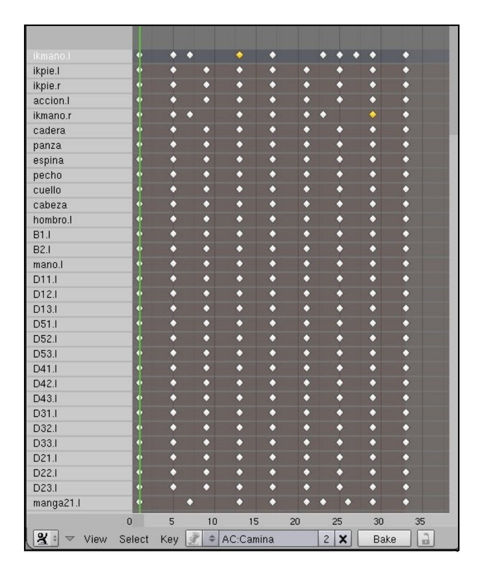


Figura 15. *Action Editor con una acción cargada.*

Las actions se visualizan y modifican en el Action Editor. Desde allí podemos crear nuevas actions, renombrarlas, modificar o borrar las keys introducidas o hacer flips de posturas para simplificar la creación de actions cíclicas.

La importancia de las Actions será crucial a la hora de exportar nuestros modelos. Éstas se exportarán, junto con la definición de la malla o separadas en otro archivo, para que el editor de niveles las cargue asignándolas a nuestro personaje.

Exportando a Doom3

Por fin estamos preparados para exportar nuestro modelo a un editor de niveles. Vamos a centrarnos, como ejemplo, en la exportación para Doom3 (por algo el título de esta charla).

Exportar personajes a md5

Doom3 tiene, como cada engine gráfico, su propio sistema para definir y reconocer las mallas de los personajes y sus animaciones. En este caso el formato es el md5mesh para la malla y sus coordenadas de textura y el md5anim para las transformaciones del esqueleto. Para ambos usaremos el script de exportación creado por Der_Ton (de los foros de Doom3world). Lo primero es conseguir el script, actualizado para la versión 2.41 de Blender, desde:

```
http://www.doom3world.org/phpbb2/➘
viewtopic.php?t=1711&postdays=0&postorder=asc&start=60
```

Antes de ejecutarlo haremos unos pequeños preparativos en nuestro personaje para que todo funcione correctamente:

- Situaremos el centro de nuestro personaje en 0,0,0
- Situaremos el centro de la Armature en 0,0,0
- Nos aseguraremos de que, almenos, hay una 3dview en los visores.
- Haremos un smooth de la malla, seleccionando el modelo, pulsando f y luego “set smooth” en los botones de edición

Una vez hecho esto abriremos el Text Editor para cargar el script, si es que no lo hemos colocado en el directorio .scripts de Blender (si lo hemos hecho podemos acceder a él directamente desde la Scripts Window) y lo ejecutaremos con Alt+p. Bastará con que definamos el tamaño, los frames de nuestra animación y el nombre de la misma y el directorio y nombre donde exportaremos el md5mesh y el md5anim.

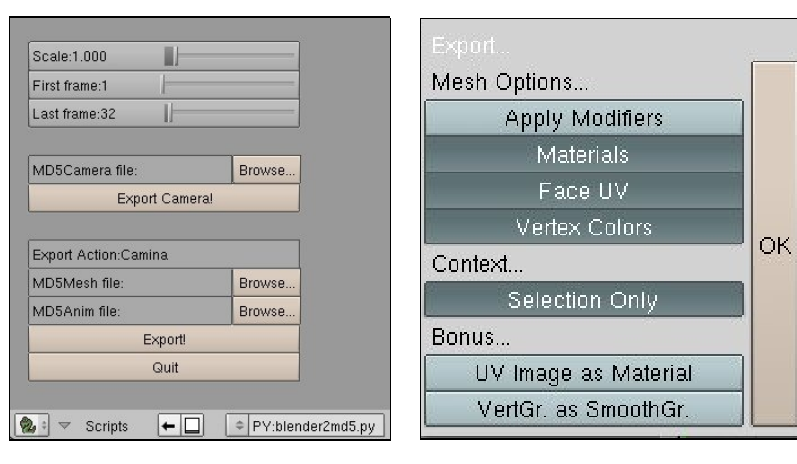


Figura 16. Izqda: *Script para exportar a MD5.* Drcha: *Exportador al formato ASE.*

Exportar objetos a ASE

Aunque lo parezca no nos hemos olvidado de los objetos. Su creación es bastante similar con la de los personajes, aunque su proceso es más sencillo por no necesitar, generalmente, animación. En el caso de crear un objeto animado deberemos tratarlo como un personaje (ejemplo: una bandera).

Normalmente sólo deberemos preocuparnos de su modelado y texturizado. Dado su menor complejidad ambas tareas deberían ser mucho más sencillas que la creación de personajes.

Para exportar objetos, que formen parte o que por si mismos constituyan un escernario, hacia Doom necesitaremos el exportador a ASE creado por **Goofos.** Dicho exportador podeis encontrarlo aquí:

```
http://www.doom3world.org/phpbb2/➘
viewtopic.php?t=9275&postdays=0&postorder=asc&start=0
```

Como con el exportador a md5 deberemos cumplir una serie de requisitos para que la exportación sea correcta:

- El modelo deberá tener coordenadas UV.
- Si el objeto tiene un material asignado deberemos activar la opción **Vcol Paint.**
- Si el objeto **no** tiene un material asignados activaremos el **VertCol**
- Asegurarnos de que la opción TexFace está activada en los botones de Editing.

Preparando para importar el personaje en DoomEdit

Empezaremos por el personaje. Tenemos los archivos que definen el personaje y sus acciones, ahora debemos crear una estructura de directorios y archivos, conectados entre sí, que sea reconocida por el Doomradiant (editor de niveles que viene con el juego original Doom3). Seguiremos los pasos siguientes:

1. Crear una estructura de directorios como la de la imagen, substituyendo el directorio llamado “geisha” por el nombre del modelo que estemos creando.
2. En el directorio **>textures >nombrepersonaje** colocaremos nuestras texturas.
3. En el directorio **>materials** crearemos el archivo **nombrepersonaje.mtr** con los parámetros siguientes:

```
nombre_de_material
{
  qer_editorimage  textures/nombrepersonaje/nombretextura.tga
  diffusemap        textures/nombrepersonaje/nombretextura.tga
  specularmap       textures/nombrepersonaje/nombretextura.tga
  bumpmap           textures/nombrepersonaje/nombretextura.tga
}
```

4. En el directorio **>models>mapaobjects>md5** colocaremos el md5mesh y el md5anim.
5. Editaremos el md5mesh colocando el nombre_de_material usado en el .mtr substituyendo el que estaba entrecomillado en la línea de **shader**.

```
mesh {
       shader "nombre_de_material"
```

6. En el directorio **>def** crearemos el archivo **nombrepersonaje.def** con los parámetros siguientes:

```
model nombrepersonaje {
  mesh  models/mapobjects/md5/nombrepersonaje.md5mesh
  anim idle  models/mapobjects/md5/nombrepersonaje.md5anim
}
entityDef nombrepersonaje {
  "inherit"  "func_animate"
  "model""nombrepersonaje"
}
```

7. Por último crearemos un archivo comprimido (zip) con los directorios **def**, **materials**, **models** y **textures**. Cambiaremos la extensión del archivo por .pk4, y lo copiaremos al directorio **>base** del Doom3.

Preparando para importar el objeto en DoomEdit

Al igual que para el personaje, para el objeto también necesitaremos una estructura de directorios parecida. Vamos a ver los pasos a seguir:

1. Crearemos los siguietnes directorios, substituyendo los llamados cubeBlender, de la imagen, con el nombre de nuestro objeto:
2. En el directorio **>textures>nombreobjeto** colocaremos nuestras texturas.
3. En el directorio **>materials** crearemos el archivo **nombreobjeto.mtr** con los parámetros siguientes:

```
textures\nombreobjeto\nombrematerial
{
  qer_editorimage   textures/nombreobjeto/nombretextura.tga
  diffusemap        textures/nombreobjeto/nombretextura.tga
  specularmap       textures/nombreobjeto/nombretextura.tga
  bumpmap           textures/nombreobjeto/nombretextura.tga
}
```

4. En el directorio **>models>nombreobjeto** colocaremos el archivo **.ASE**.
5. Editaremos la ruta hacia las texturas del archivo **.ASE** para que sea relativa.
6. Por último crearemos un archivo comprimido (zip) con los directorios **materials**, **models** y **textures**. Cambiaremos la extensión del archivo por .pk4, y lo copiaremos al directorio **>base** del Doom3.

Abriendo el DoomEdit

Podríamos usar el GTKRadiant, aunque ahora usaremos el DoomEditor que viene con el juego. Para abrirlo podemos crear un acceso directo con el siguiente path, respetando las comillas y el **+editor**

```
Directorio_instalación>doom3.exe” +editor
```

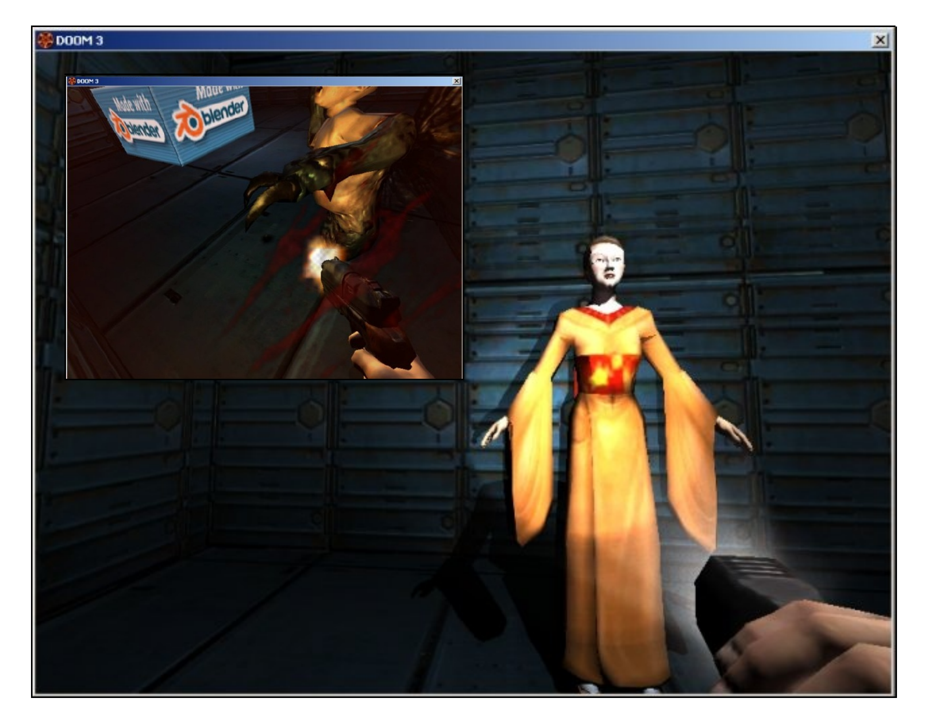


Figura 17. *Capturas de Doom3 con un personaje y un objeto creados en Blender.*

El uso de este editor podría llenar el contenido de toda una party, así que dejaré un enlace web para que se pueda investigar a partir de ahí, en la siguiente URL: `http://www.modwiki.net/wiki/DoomEdit`

De momento saber que ya podemos cargar nuestros modelos pulsando con el botón derecho del ratón sobre el espacio de construcción ;-)

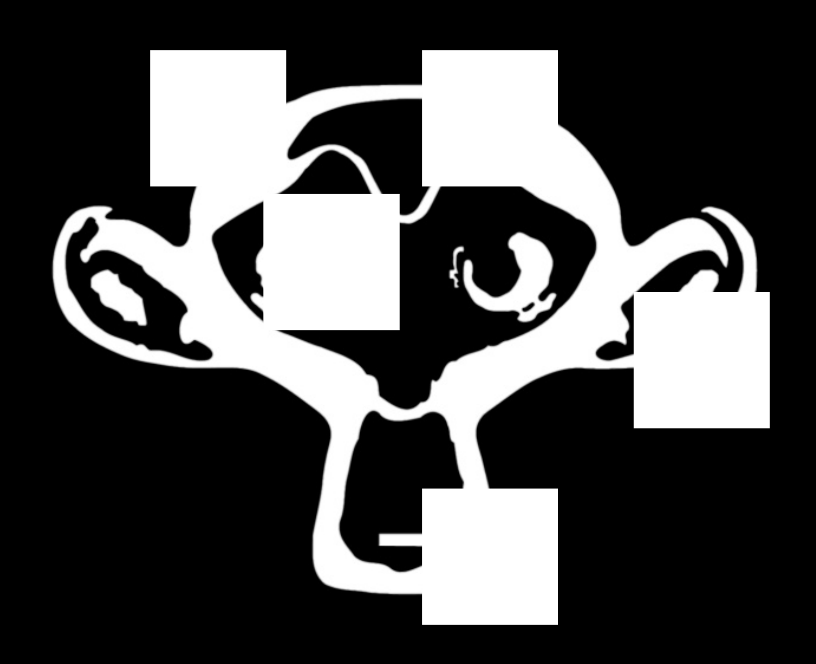
Saltar al desarrollo Indie

Hemos visto, de forma muy general, que tenemos la posiblidad de crear personajes, con Blender, para incluirlos en un mod, pack o mapa del Doom3. La utilización de scripts de exportación permiten, hoy día, crear contenidos gráficos, con Blender, para motores que son muy usados por la comunidad Indie para la creación de videojuegos. Quizás los más representativos podrían ser el Torque o el Blitz3D (muy asequibles), y el Ogre o el Irrlicht (GPL). Obviamente para el uso de estos motores ya se necesita un equipo de desarrollo, o como mínimo la presencia de al menos un programador y un grafista.

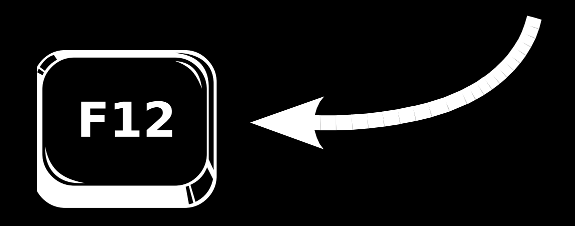
Sección IV
Render

Fotograma de la Película de Animación
"Elephants Dream" realizada con Blender





Rendering or...
how to draw a *monkey*



Iluminando con Yafray

Alejandro Conty Estévez
Fernando Arroba Rubio
aconty@gmail.com - gnotxor@gmail.com ::



En esta exposición vamos a ver cómo se comporta YafRay a la hora de realizar un render y cómo podemos controlarlo desde Blender. Para ello trabajaremos siempre en modo "full" y describiremos cada uno de los controles que soporta. Es decir, partiendo de que en método "full" se tienen en cuenta tanto la luz directa como la indirecta, nos encontramos con dos bloques de controles para ajustar esa iluminación: la "cache" y los "fotones". Abundaremos, por tanto, en cómo funcionan estos dos elementos.

Para entender mejor el funcionamiento de YafRay nos detendremos un poco en cómo se definen las luces y más concretamente en cómo influye dicha definición en la aparición de "ruidos" y otro tipo de problemas en la escena. Por supuesto, también veremos cómo podemos hacer frente a éste tipo de fenómenos y las mejores opciones para conseguir buenos resultados en tiempos de render aceptables.

¡Hágase la luz!

Un *raytracer* intenta simular el comportamiento real de la luz cuando genera las imágenes. Esta simulación debe equilibrarse entre la calidad de la imagen y el tiempo de render. Por tanto, está también limitada a la máquina sobre la que esté trabajando y otros factores. Por ejemplo, sería imposible lanzar rayos desde los objetos luminosos en infinitas direcciones y calcular su influencia con sus posibles rebotes, teniendo en cuenta que sólo son útiles esos cálculos en los pocos casos que estos interactuaran con la "cámara". Así pues, la simulación se hace desde el concepto contrario, averiguar qué intensidad y matiz tiene un punto desde la posición de la cámara. Lo que implica algoritmos diferentes según el tipo de luz, con sus ventajas e inconvenientes.

Cuando el origen de la luz es una fuente puntual se emplearán unos algoritmos locales, rápidos de calcular y absolutamente predecibles. Sin embargo, este tipo de iluminación es menos realista, produce bordes duros

en las sombras y la luz que producen estará poco matizada por rebotes.

Otros tipos de luces son superficies, son más realistas y producen una sombra y penumbra mucho más matizadas. Sin embargo, el problema de este tipo de luces es la dificultad de cálculo. Los métodos que se emplean son "estocásticos", también llamados de "montecarlo" o de "muestreo al azar". Este método es lento pues consiste en tomar muestras del entorno disparando rayos en todas las direcciones para calcular una media de intensidad y matiz del color.

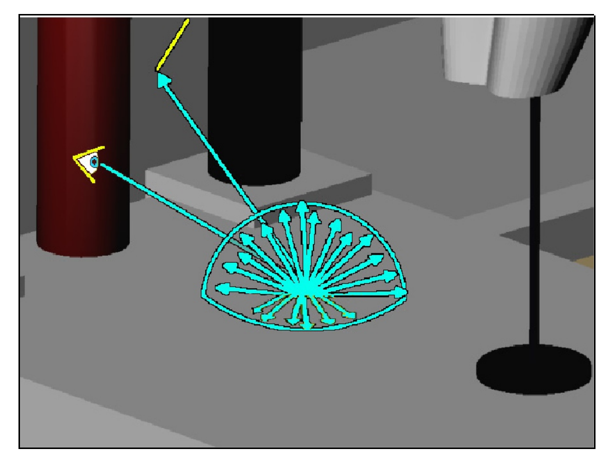


Figura 1. *Método de Montecarlo*

Como se puede ver representado en la figura 1, técnicas de montecarlo lo que hacen es un muestreo aleatorio del entorno, de manera similar al *Ambient Occlusion*, que captura la influencia de los posibles rebotes de luz del ambiente.

La luz que ilumina una escena no solo procede de "fuentes de luz" propiamente dichas. En la realidad podemos encontrarnos distintos tipos de luz: a) la que emite el cielo - no solo el sol de forma directa - y que la podemos simular mediante una imagen de fondo - sea HDR o no –; b) la que pueden emitir distintos objetos del entorno; y c) la que devuelven otros objetos en los que rebota la luz. Para nuestro caso, todas estas luces se tratarían como "luces no puntuales" y se calcularían por el método montecarlo.

Como se puede observar en la figura 2, la iluminación que proviene de todas partes del cielo puede producir una escena muy iluminada sin apenas matices de sombra. En este caso es el cielo un gran emisor de luz.

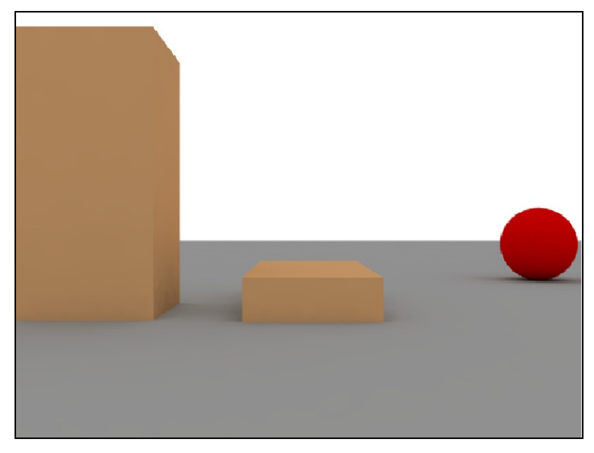


Figura 2. *Luz que proviene del fondo.*

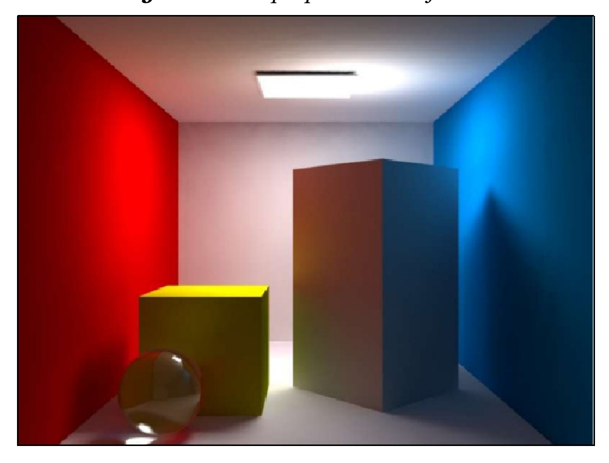


Figura 3. *Luz proveniente de un objeto.*

En el caso de la figura 3 podemos observar que la iluminación está mucho más matizada que en el caso anterior. Hay más contraste entre las zonas iluminadas y las zonas en sombra y entre ambas existe una zona de penumbra con muchos matices. Además hay que resaltar los matices de color que se aportan entre sí los distintos objetos.

En este último caso, representado por la figura 4 podemos observar, igual que en la figura 3, que la iluminación no proviene de una fuente de luz puntual sino de una zona más amplia. Al contrario que en el caso anterior, la iluminación proviene de una zona intensamente iluminada de la habitación. En último caso sí hay una fuente de luz pero la “iluminación” es indirecta, se basa en el rebote de la luz en las superficies.



Figura 4. *Luz proveniente de una zona iluminada.*

El ruido

El *ruido* es un problema que aparece con cierta frecuencia en determinadas escenas. Son una serie de manchas o contrastes de luz y sombra que se acentúan bajo algunas condiciones. La condición ideal para que aparezca el ruido es la utilización de fuentes pequeñas y potentes. El porqué aparece es fácil de entender: si la fuente de luz es pequeña es menos probable de encontrar mediante el trazado de rayos aleatorio de las técnicas montecarlo. Si un punto determinado no encuentra la fuente de luz y el punto contiguo sí, aparecerá un contraste entre ellos (uno iluminado y el otro en sombra); dicho contraste será más acentuado si la luz es potente.

Cuanto más pequeña sea la fuente de luz, más improbable será encontrarla por “montecarlo”. Cuanto menor sea la probabilidad de encontrar la luz, más intenso será el ruido.

Los casos típicos en los que aparecen ruidos son tres: 1) un objeto emisor muy pequeño (como se ha dicho antes); 2) una ventana muy pequeña (por idénticos motivos) y 3) un HDR con la luz muy concentrada.

En realidad el problema no es el tamaño de la fuente de luz sino la visibilidad de ésta desde los distintos puntos de la escena. Es decir, una fuente de luz lejana es más difícil de encontrar trazando rayos de manera aleatoria, es decir es menos visible; o, si se prefiere, el angulo visto desde el punto es más estrecho, como se aprecia a simple vista en la figura 5.

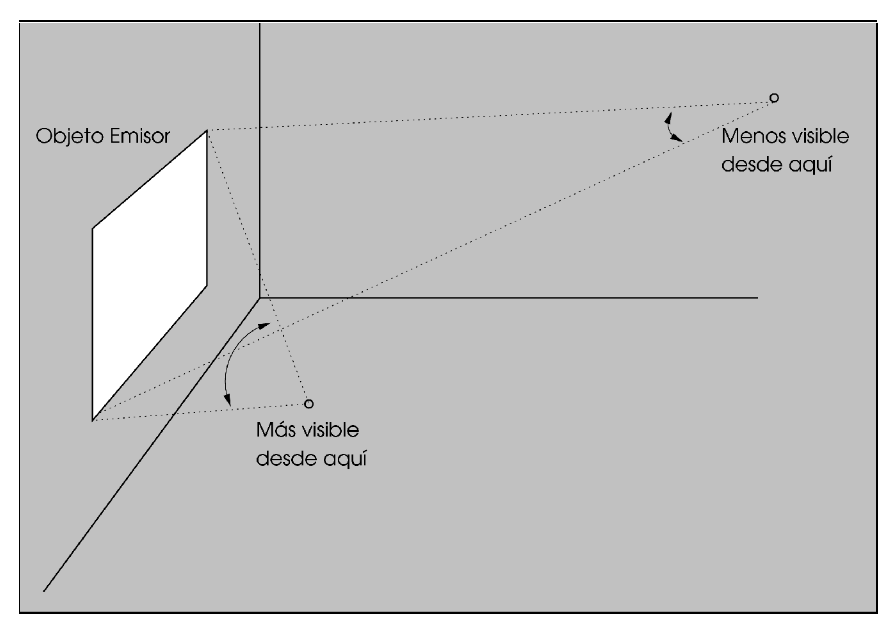


Figura 5. *La visibilidad: el verdadero problema.*

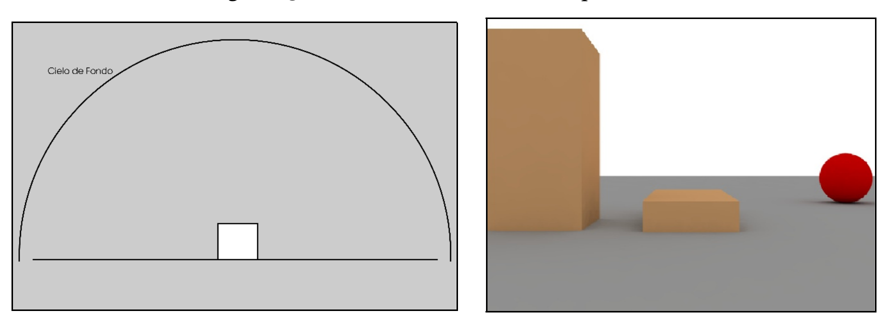
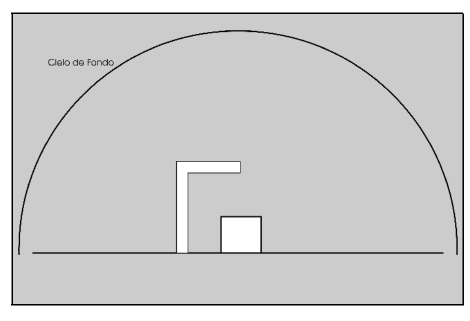


Figura 6. Izqda: *El cielo es totalmente visible. Es fácil acertar en la fuente de luz con un rayo.* Drcha: *No hay sombras muy marcadas ni contrastes, ni aparecen manchas ni ruido en la imagen.*

Vamos a ver cómo afecta la visibilidad de la fuente de luz a la producción de ruido. Comenzaremos por una situación irreal en la que cualquier rayo encontrará la fuente de iluminación, en nuestro caso el cielo.

Ahora vamos a tapar la mitad del cielo. Eso dará como resultado que sólo el "50%" de los rayos alcancen la fuente de luz. Se puede apreciar cierto ruido en el render fruto de los contrastes entre los rayos que encuentran la fuente de luz (el cielo) y los que no, que aproximadamente serán la mitad de los calculados.



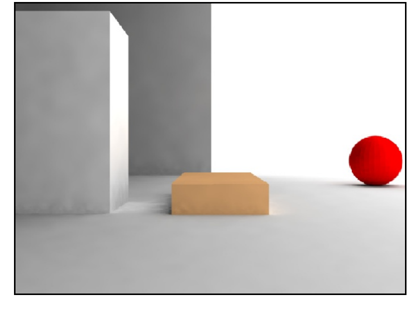
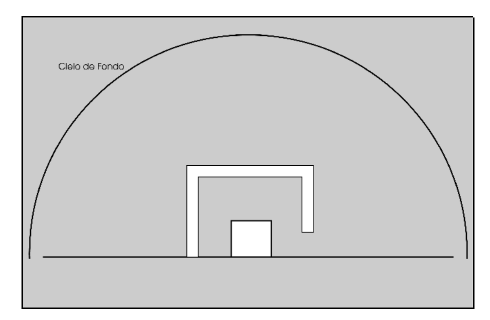


Figura 7. *a) El cielo está parcialmente oculto. b) Aparece un poco de ruido en el render.*



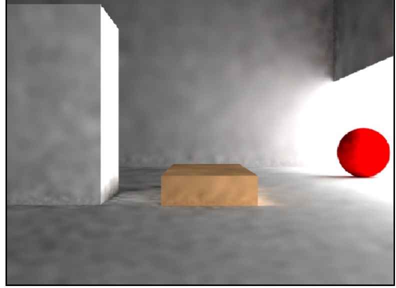


Figura 8. *a) Cielo poco visible. b) Aparece mucho ruido.*

Si reducimos la proción de cielo visto, por ejemplo desde una ventana la iluminación decrecerá, pues se ha restringido el número de rayos que interceptarán la fuente de luz.

Para compensar la disminución de la luz, se ha incrementado la potencia de la luz 8 veces.

El ruido que aparece en la escena porque la fuente de luz es pequeña se ve intensificado porque ha habido que incrementar la potencia de la luz para conseguir la luminosidad deseada. Además, las zonas que se encuentran iluminadas directamente por la fuente de luz aparecen quemadas.

Reduciendo el ruido

Cuando aparece este ruido intentamos reducirlo de la manera más obvia. Subimos la "*Quality*" en el panel de YafRay. Esto soluciona algunos casos, pues hay que tener en cuenta que dicho control incrementa el número de "*samples*". Un mayor número de samples incrementa la probabilidad de encontrar una luz cuando esta es pequeña, sin embargo hay un problema: el tiempo de render se incrementa notablemente. Mientras el tiempo de

render aumenta linealmente, el ruido decrecerá en $1/\sqrt{samples}$. Por lo tanto habría que utilizar este método con cuidado y ayudarse con los fotones y con el "*refinement*" de la caché.

Qué hacer cuando las luces son pequeñas

Normalmente el problema de tener fuentes de luz pequeñas se suele producir en escenas interiores donde la luz puede provenir de pequeñas fuentes de luz artificial o de ventanas. Para ayudarnos en este tipo de render podemos utilizar los fotones. No hay que confundir esto con los fotones que utilizamos para generar cáusticas.

Básicamente lo que hacemos con el uso de un buen mapa de fotones es darle información extra al método montecarlo, como de donde proviene la luz o estimaciones de luz indirecta, con lo que además nos ahorramos los rebotes.

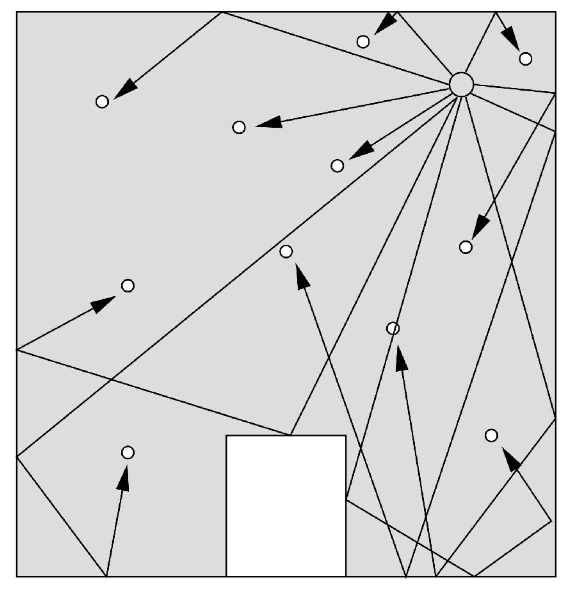


Figura 9. *Lanzamos fotones desde las fuentes de luz y rebotan dejando marcas, con esas marcas hacemos el mapa de fotones.*

El mapa de fotones se puede obtener visualmente teniendo seleccionado el botón "*Tune Photons*" en el panel GI de YafRay. Este mapa es una aproximación a cómo quedará el render final. Es muy recomendable acostumbrarse cuando iluminamos a trazar este mapa para la escena, pues nos aportará información valiosísima para conseguir la distribución de luz que queremos.



Figura 10. *Ejemplo de mapa de fotones.*

Configurar los fotones

Para configurar los fotones debemos seguir unos pasos muy sencillos. Lo primero de todo es activar el botón fotones en el panel de YafRay. Lo siguiente es escoger un número y un radio.

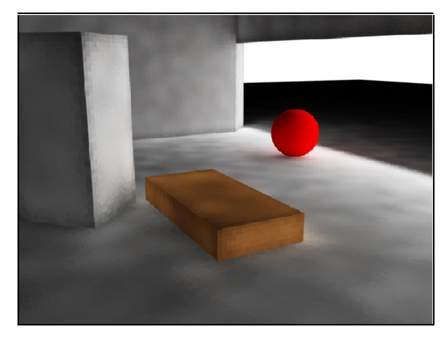
En este punto hay que hacer algunas consideraciones, pues cuantos más fotones lancemos más preciso será el mapa, sin embargo también aumentará el tiempo de cálculo. Cuantos más fotones empleemos más precisión y menos manchas o ruido tendremos en nuestra escena. En cuanto al radio, cuanto más grandes son más suave será el mapa de fotones, pero eso no debe engañarnos pues también estaremos siendo menos precisos.

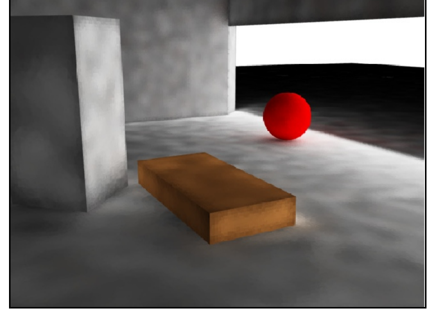
¿Cuántos fotones usar y con qué radio? Pues depende fundamentalmente de dos parámetros. Uno es el tamaño físico de la escena en unidades, no en complejidad. Otro es la precisión que queremos alcanzar.

Como hemos dicho los parámetros óptimos dependen de la escena concreta y por tanto no se pueden dar recetas mágicas sobre cómo hacerlos. Pero al menos sí sabemos cómo debe ser un buen mapa de fotones. Debe tener dos características fáciles de entender:

1. No tener *light leaks* o “agujeros de luz” visibles. Es decir, zonas de luz que aparecen en sitios que se suponen deberían estar en penumbra. Esto suele suceder cuando el radio de los fotones es demasiado grande para la escena y sobrepasa notablemente el tamaño de los objetos, por lo que los fotones pueden estar atravesandolos.

2. Debe ser tan suavizado como sea posible. Recordamos que si utilizamos un radio de fotón demasiado grande el mapa quedará suavizado pero estaremos perdiendo precisión en la iluminación. Un buen ejemplo de mapa de fotones suavizado es el de la figura 10.





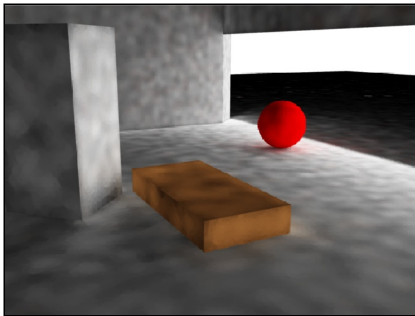


Figura 11.

Arriba-Izq: *Mapa de fotones con radio 1.0*
Arriba-Dch: *Mapa de fotones con radio 0.7*
Abajo-Izq: *Mapa de fotones con radio 0.4*

En otras palabras, debemos utilizar un radio de fotón lo más grande posible para que quede suave pero cuidando de que no aparezcan *light leaks*.

El tener un buen mapa de fotones es fundamental para el ahorro de tiempo de render. Cuando YafRay se dispone a trabajar con las técnicas de *montecarlo* lanzando rayos de forma aleatoria, encuentra que existe un mapa de fotones. En lugar de ponerse a calcular rebotes para encontrar las fuentes de luz, lee ese mapa de fotones que ya tiene calculados la luminosidad y color de la zona. Con esos datos no necesita continuar con los cálculos, ni saber dónde están situadas las luces.



Figura 12. *El resultado del mapa de fotones de la figura 10.*

Casos comunes donde pueden ayudar los fotones

Nos podemos encontrar con varios casos en los que los fotones nos pueden ser de gran ayuda. Por ejemplo cuando necesitamos pequeñas zonas iluminadas en la escena podemos utilizar luces tipo *spot* y *omni* como emisores de fotones. Cuando necesitamos simular que el emisor es un objeto podemos utilizar luces de área y configurarlas como emisores de fotones. Igual podemos utilizar las luces de área si lo que nos interesa es simular la entrada de luz por una ventana.

Pequeñas zonas iluminadas: estudio de dos casos.

El primer caso es cuando tenemos una ventana por la que entra el sol. En este caso se iluminará directamente una pequeña área de la habitación, sin embargo el resto de la habitación debería estar iluminada de forma indirecta.

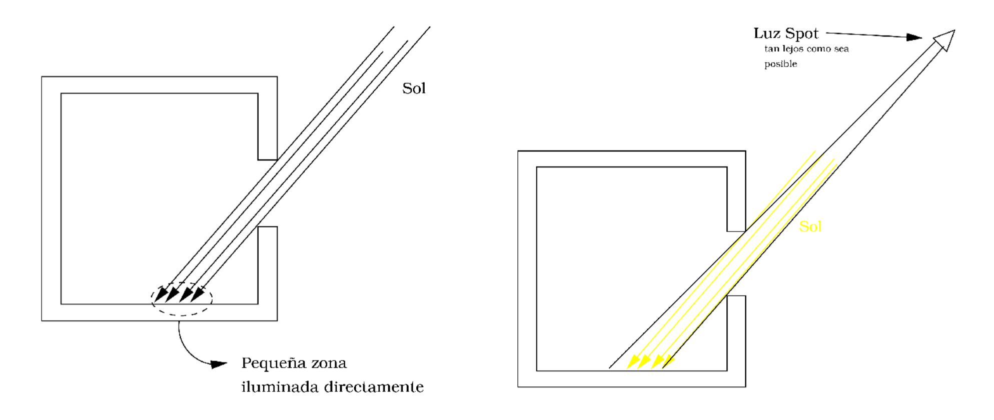


Figura 13. Izqda: *El sol entrando por una ventana.*
Drcha: *Sustituyendo el sol por una spot.*

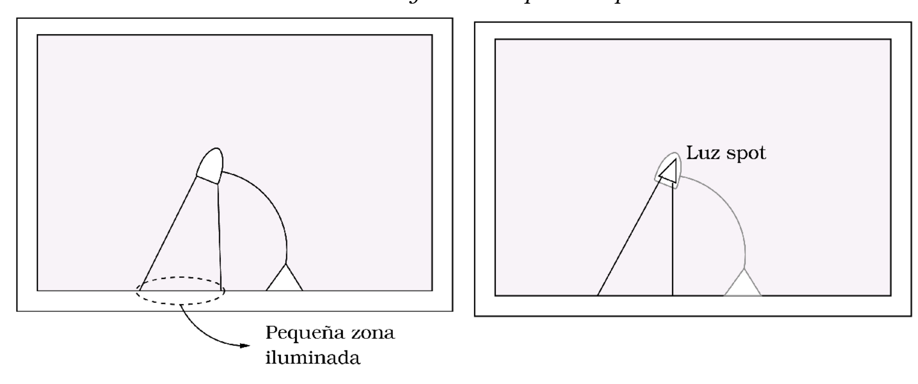


Figura 14. Izqda: *Iluminación interior* · Drcha: *Iluminación interior con spots.*

En la figura 13 (Izqda) podemos apreciar un esquema de ilustrativo de este caso. Para simularlo podemos emplear el método descrito en la figura 13 (Drcha).

El otro caso que vemos es el de iluminación interior. En este caso el problema es muy similar: aparece una pequeña zona iluminada directamente. Como en el caso anterior utilizaremos luces tipo *spot* o incluso *omni*, pues en este supuesto los fotones no escaparán de la escena.

Como en el caso anterior deberemos colocar cada luz en el sitio natural de la escena. Si tenemos luces con pantallas como en la figura 14, es aconsejable utilizar *spots* para simularlas. Sin embargo, es mejor utilizar luces de tipo *omni* en el supuesto de que sean puntos de luz sin apantallar como es el caso de muchas lámparas cuyas bombillas se encuentran al aire.

Objetos emisores

En otras ocasiones nos encontramos con objetos que emiten luz, algunos de geometrías complicadas. Estos objetos pueden ser desde simples plafones a complicados anuncios de neón. Simularlos puede ser por tanto muy sencillo o muy trabajoso dependiendo de esos factores.

Identificar dichos objetos en una escena 3D es sencillo, son todos los objetos cuyo *emit* esté activado. Para simularlos emplearemos una o más luces de área. Lo ideal sería conseguir una luz de área que se ajustase al objeto lo más fielmente posible, por lo tanto: si el objeto es cuadrado nos bastará con una, si es más complejo tendremos que utilizar varias para cubrir toda la superficie del objeto lo mejor posible.

Por supuesto, el color de la luz emitida por dichas áreas debe coincidir con el del objeto. Y por último, el *power* debe estar fijado a 1.0. Tenemos la tendencia de subir el *power* cuando encontramos zonas mal iluminadas en el render. Esto produce un efecto de quemado en las zonas cercanas a la luz pero sigue manteniendo las otras zonas más oscuras. Si utilizando los fotones y sus rebotes no mejora la iluminación, debemos comprobar que la luz no se encuentre bloqueada por algún objeto o, incluso, agregar iluminación de relleno.

Para simular objetos emisores emplearemos, como ya se ha dicho, al menos una luz de área, que deberemos situar lo más cerca posible del objeto emisor pero cuidando que no se solape con el mismo. Si se solapa pueden aparecer efectos extraños o incluso no tener ningún efecto si los fotones quedan bloqueados por el objeto emisor.

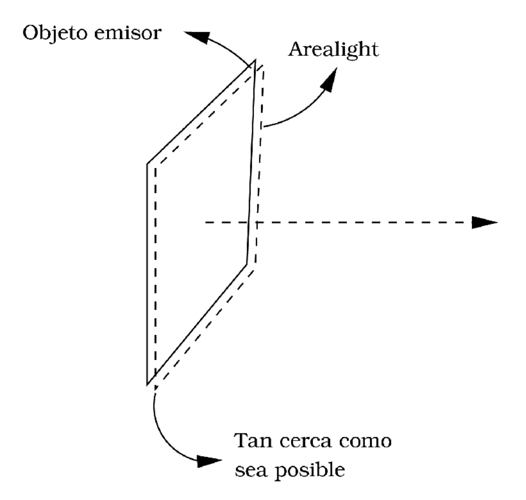


Figura 15. *Arealight cubriendo un objeto emisor.*

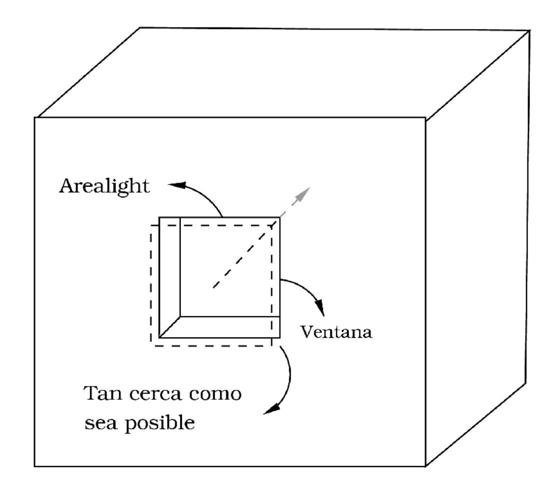


Figura 16. *Cubriendo una ventana con una luz de área.*

Luz difusa por una ventana

Otro caso similar es la entrada de luz difusa procedente del cielo desde una ventana. Como en los casos anteriores utilizaremos al menos una luz de área para cubrir la ventana. Bastará con una si la ventana es sencilla, aunque tendremos que utilizar varias para ventanas complejas, aunque una simple aproximación será suficente.

Como en los casos anteriores, debemos situar la luz de área lo más cerca posible de la ventana, apuntando hacia adentro, cubriendo toda la superficie pero a la vez intentando que se escapen el menor número de fotones posible. El color del área debe ser el mismo que presente el cielo y el *power* debe estar fijado a 1.

¿Y ahora qué?...

Bueno, hasta aquí hemos visto cómo utilizar distintos tipos de luz para simular fuentes de fotones. Vamos ahora a pensar como el usuario medio... Supongamos que hemos realizado una escena donde las fuentes de luz son pequeñas y queremos optimizar el render.

Haciendo caso de lo visto hasta ahora decidimos que vamos a utilizar los fotones. Así pues, nos aseguramos de que todas las fuentes de luz pueden emitir fotones cambiando las que sean necesarias por *spot* u *omni*, y colocando luces de área para simular objetos emisores, ventanas...

También hemos configurado un número de fotones y un radio racionales (500.000 y 0,5 para empezar, por ejemplo). Una vez hecho todo esto, activamos el botón *tune photons* para conseguir el famoso mapa de fotones... ¿y ahora qué hacemos?

Pues lo primero será comparar el mapa de fotones que hemos obtenido con un render de baja calidad. Si tienen la misma intensidad estamos en el buen camino, pero si no coincide tendremos que revisar nuestras luces para comprobar que no nos hemos olvidado de activar algún color en alguna fuente de luz (también podría ser un bug de YafRay). Si siguen apareciendo *light leaks* puede ser debido a un modelado no sólido, por lo que tendríamos que revisar la geometría de nuestra escena.

Hasta aquí está todo más o menos claro, pero ... ¿Se puede refinar aún más la imagen? Pues sí, vamos a verlo.

Irradiance Cache

Para refinar un poco más la imagen contamos con la ayuda del *irradiance cache*. Para poder utilizarlo correctamente vamos a ver primero cómo funciona.

El *irradiance cache* evita tener que calcular la iluminación global en cada pixel. Básicamente lo que hace es seleccionar algunos pixeles para realizar los cálculos y luego interpola. Esta interpolación disolverá un poco el ruido, sin embargo, también puede producir ruido de baja frecuencia (nubes). En animaciones, esta interpolación produce parpadeos bastante molestos, aunque funciona bien y reduce el tiempo de render en imágenes estáticas.

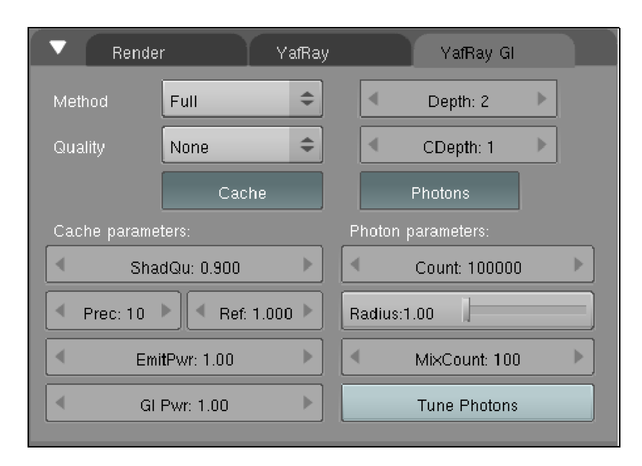


Figura 17. *Configuración del cache.*

Para activar el cache necesitamos tener seleccionado el método *full* y activar el botón *Cache* para acceder a todos los parámetros de configuración. Para entender mejor como funciona vamos a ver las etapas del render:

1. **Fase de disparo de fotones.** Desde cada emisor potencial que haya en la escena se comienzan a lanzar los fotones; repartidos equitativamente entre los emisores.
2. **Primer pase de render** (*fake pass*). El parámetro "*Prec*" determina cada cuántos pixels calculará la cantidad de luz, aunque en las esquinas tomará más muestras.
3. **Etapa de refinamiento** (2 *fake passes* más). En este paso identifica cambios bruscos de luz que superen el valor determinado por *Refinement*. Cuando encuentra valores superiores al refinado lo que hace el programa es realizar un "*anti-alias*" de la luz tomando más muestras.
4. **Fase final**. Interpola con el cache resultante y genera la imagen.

Ahora que comprendemos mejor los pasos del render, vamos a detallar un poquito más los parámetros y su funcionamiento.

Por ejemplo, la calidad de las sombras. *Shadow quality* debe ser un valor entre 0 y 1, aunque el valor ajustado por defecto es 0.900 que ha demostrado servir para casi cualquier escena. Lo que hace este parámetro es controlar como crece la densidad de muestras en las esquinas; por tanto, a más muestras, a un valor más alto, mayor definición de las sombras en las esquinas. Por lo tanto, teniendo ajustado el valor por defecto, es mejor dejar que sea el refinamiento el que mejore el resultado.

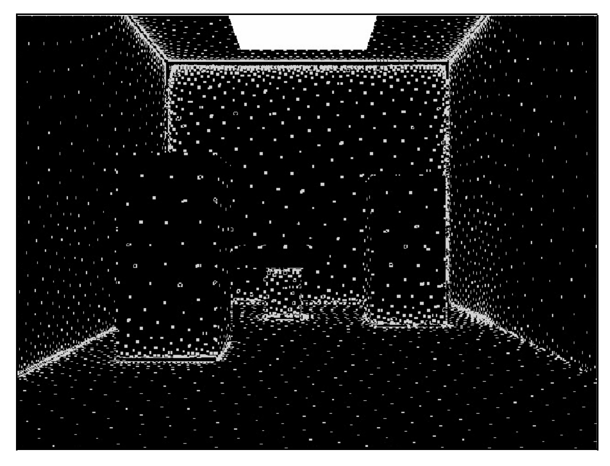
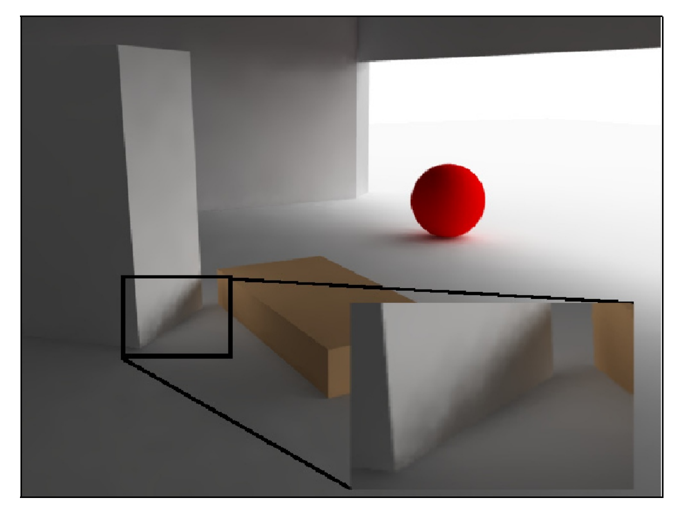


Figura 18. *Distribución de muestras.*

Para utilizar el *Refinement* tenemos un valor entre 0 y 1 también. Como ya se adelantó antes al ver los pasos del render, este parámetro determina qué cambios de luz serán refinados. Cuando se produce un cambio de luz que supera el valor establecido, como en los bordes de las sombras o los artefactos debidos a bajo número de muestras, se realiza el refinado. En este caso, el pase se repite dos veces. Si establecemos el valor de *refinement* a 1, estaremos desactivando el refinamiento, mientras que si lo establecemos a 0.03, por ejemplo, el refinamiento será exhaustivo.



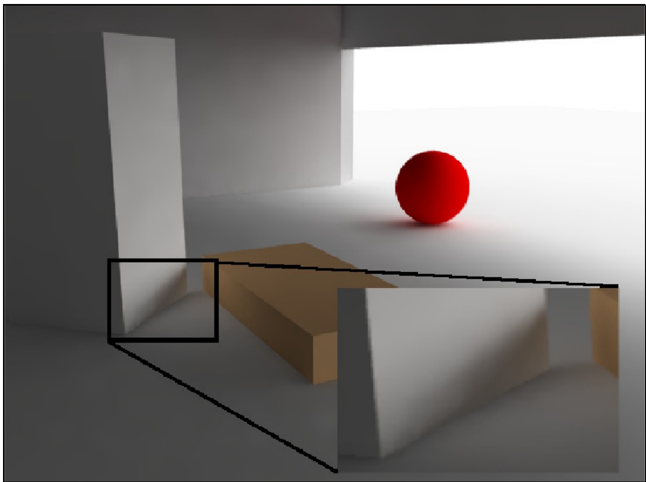


Figura 19. Arriba: *Render original, sin refinamiento.*
Abajo: *Render con refinamiento a 0,05.*

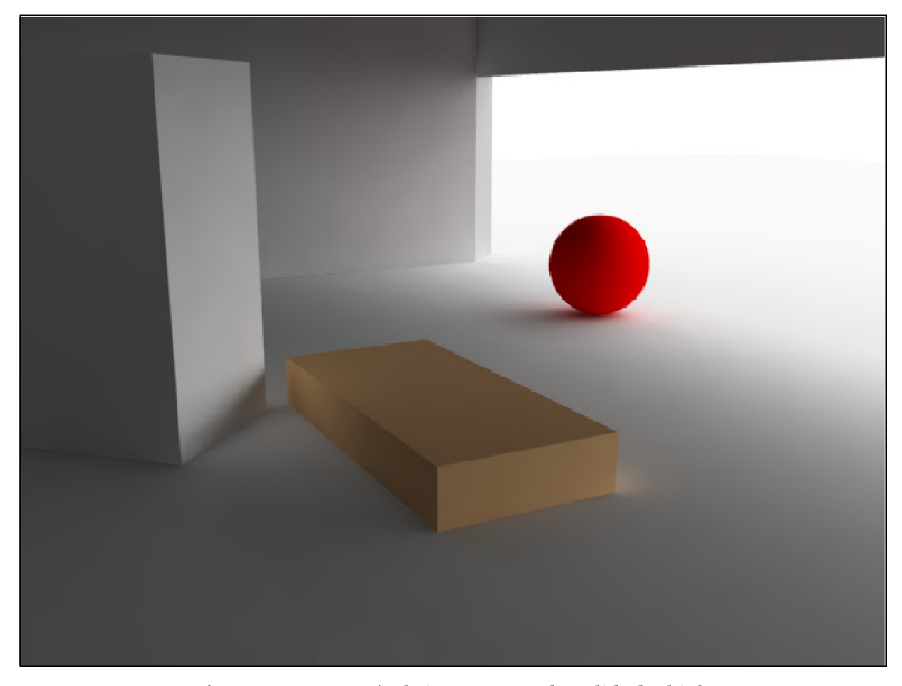


Figura 20. *Después de incrementar la calidad a high.*

Hemos conseguido que las sobras estén más matizadas, más suavizadas, pero aún queda algo de “suciedad” por la escena, aunque ya no está situada en los bordes de las sombras. ¿Qué podemos hacer para “limpiar” la escena? Pues se nos pueden ocurrir dos cosas: la primera es suavizar el mapa de fotones y la otra es subir los samples (*Quality*).

Lo de suavizar el mapa de fotones tampoco nos servirá de mucho, pues pasado un cierto punto, o dicho de otro modo, suavizado en exceso, aparecerán “*light leaks*”.

Cáusticas

Hay dos manera de conseguir cáusticas con yafray, desde una luz puntual o desde un objeto, cielo o fuente no puntual.

Cuando las cáusticas las queremos conseguir desde una fuente puntual es imprescindible que utilicemos una “*photonlight*” explícita.

Si las cáusticas las estamos trabajando con fuentes de luz no puntuales no es necesario utilizar una *photonlight*, sino la forma de trabajo con fotones que se ha descrito anteriormente con algunas matizaciones. Trabajando en

el método *full* deberíamos conseguir las cáusticas sin ningún tipo de problema. Sin embargo, es recomendable que las fuentes de luz no sean muy pequeñas. Además tendremos que vigilar el parámetro *CDepth* y aumentarlo si fuera preciso.

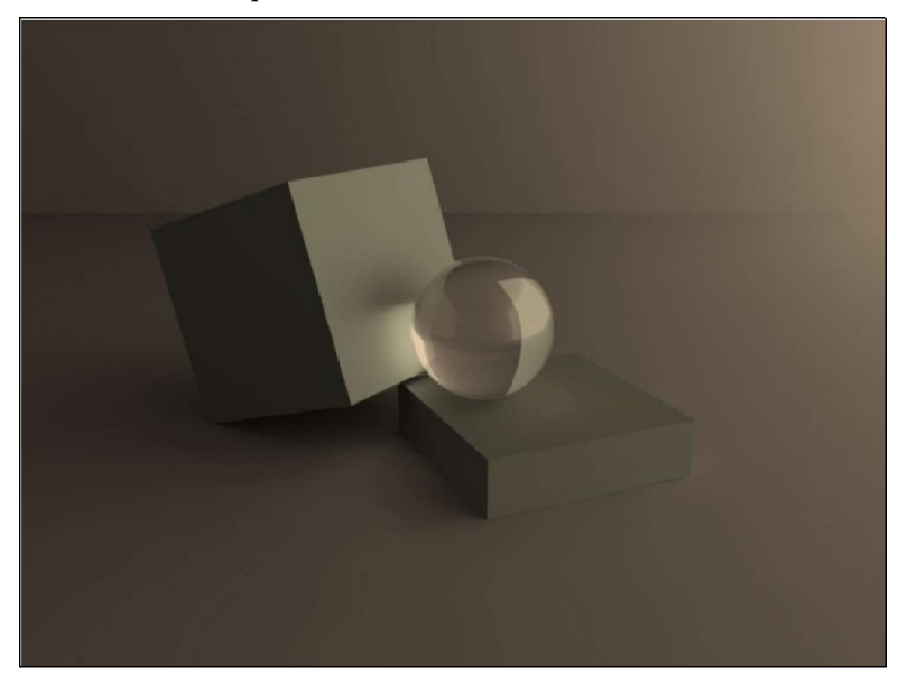


Figura 21. *Causticas desde una ventana.*

YafRid: Sistema Grid para render

Jose Antonio Fernández Sorribes
Carlos González Morcillo
josean_fdez_sorribes@yahoo.es ::



Q El último paso en el proceso para la generacióon de imágenes y animaciones 3D por ordenador es el llamado render. En esta fase se genera una imagen bidimensional (o un conjunto de imágenes en el caso de las animaciones) a partir de la descripción de una escena 3D. Para la obtención de imágenes fotorrealistas se utilizan algoritmos computacionalmente exigentes como el trazado de rayos.

En los proyectos relacionados con la "síntesis de imágenes", la etapa de render se suele considerar el cuello de botella debido al tiempo necesario para llevarla a cabo. Generalmente, el problema se resuelve usando granjas de render que pertenecen a empresas del sector en las que los frames de una animación se distribuyen en distintos ordenadores. El presente proyecto ofrece una alternativa para acometer el render tanto de animaciones como de imagenes basada en computación grid. Para ello se ha desarrollado un sistema en el que tienen cabida ordenadores heterogéneos tanto en software como en hardware, distribuidos geográficamente y conectados al grid via internet.

Objetivos

El objetivo principal del presente proyecto es el de construir un prototipo funcional de un sistema se render distribuido basado en computación grid optimizando el tiempo necesario para llevar a cabo el proceso. Además de este objetivo fundamental, existe una serie de objetivos complementarios que son descritos a continuación.

Arquitectura multiplataforma

Este sistema debe estar desarrollado sobre una arquitectura que permita su funcionamiento en distintos sistemas operativos y con independiente del hardware. Esto permitirá tener un grid heterogéneo en todos los sentidos ya que cualquier ordenador con conexión a internet es un potencial proveedor de servicio. Es en este aspecto donde se aleja de los clásicos clusters de ordenadores.

Independencia del motor de render

Aunque el sistema se orientará a su utilización con dos motores de render en concreto (Yafray y el que posee Blender 3D), el desarrollo se debe realizar de forma que sea posible añadir nuevos motores de render sin excesivo esfuerzo.

Varias granularidadades

El sistema desarrollado debe permitir la distribución del render tanto de imágenes como de animaciones completas. Las unidades mínimas de distribución, denominadas unidades de trabajo, serán el fragmento de frame y el frame respectivamente.

Interfaz web

La mayor parte de la funcionalidad del sistema debe ser controlable via web. Esto aporta una ventaja clave para ser realmente utilizado por la comunidad de usuarios ya que no es necesario instalar ningún software, cualquier navegador es suficiente.

Grupos de Usuarios

Se debe poder permitir la creación de grupos de usuarios (clientes y proveedores) con el fin de ajustar qué proveedores atienden a qué proyectos. Mediante esta utilidad se pueden establecer subgrupos privados de proveedores que proporcionan servicio a los proyectos de determinados clientes.

Prioridades

Se establecerá además una sistema de prioridades de los proveedores y los clientes con respecto a los grupos a los que pertenecen.

Diseño

Un adecuado diseño del sistema permitirá que en un futuro se añadan nuevas funcionalidades o se mejoren las existentes de una forma sencilla. El uso de una división en capas y patrones de diseño así como la existencia de una completa documentación facilitarán futuros desarrollos.

Usabilidad

Tanto el software del proveedor como el interfaz web que son los dos medios principales a través de los cuales el usuario interactúa con el sistema deben ser fáciles de usar. Se tratará de realizar interfaces cómodas e intuitivas.

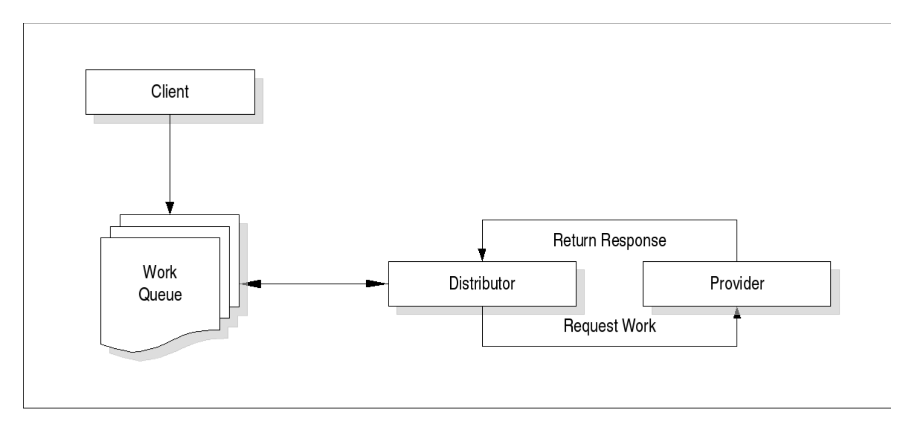


Figura 1. *Ciclo de interacción en un grid computacional.*

Instalación sencilla de proveedor

El software necesario para que un usuario se convierta en proveedor y pase a formar parte del grid debe ser sencillo de instalar y configurar en las distintas plataformas disponibles. Se intentará reducir el tiempo dedicado a estas dos tareas al mínimo posible.

Inicio automático

A fin de que colaborar con el grid requiera el menor esfuerzo posible, se ha de configurar el software de modo que al iniciar el sistema, se inicie el software para que el usuario no tenga que estar pendiente.

Arquitectura

El tipo de grid más adecuado para implementar la distribución del render de una escena entre múltiples máquinas es el **grid computacional**. Esto se deduce de las características del render, caracterizado por ser un proceso muy intensivo en términos de consumo de CPU.

Conceptualmente, los componentes mínimos de un grid computacional son los siguientes:

- **Distribuidor.** Constituye el *corazón* del grid. Su principal tarea es la de tomar los trabajos de la cola y distribuirlos entre los proveedores de la forma más conveniente posible.
- **Proveedor de servicio.** Esta entidad es responsable del procesamiento real de las peticiones de los clientes.

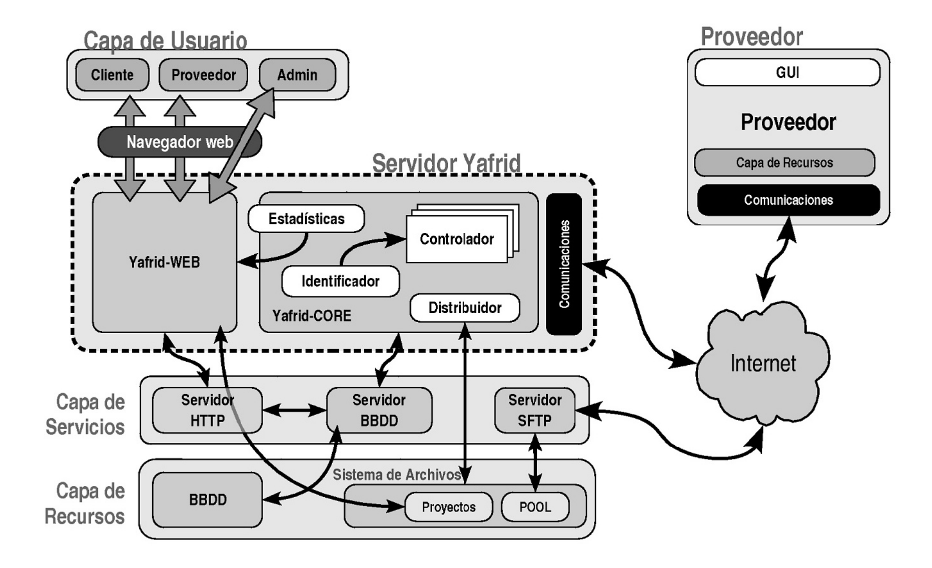


Figura 2. *Arquitectura general del sistema Yafrid.*

- **Cliente.** Un cliente es una entidad externa que no pertenece al sistema en un sentido estricto. Su papel en el funcionamiento del grid pasa por enviar trabajos al sistema para ser realizados por los proveedores. Estos trabajos son almacenados en una cola de la cual el distribuidor irá elegiendo las próximas peticiones en ser procesadas.

Estos componentes interactúan en un ciclo que se muestra esquemáticamente en la Figura 1.

La arquitectura desarrollada para soportar el sistema distribuido de render Yafrid se muestra en la Figura 2. A continuación se realiza una introducción general a esta arquitectura cuyos componentes serán ampliamente analizados en sus respectivas secciones.

Servidor Yafrid

El Servidor Yafrid es el nodo fundamental alrededor del cual se establece el sistema de render Yafrid. Cada uno de los proveedores se conecta a este nodo para hacer que sus ciclos de CPU puedan ser usados para renderizar las escenas enviadas al grid por los clientes.

El Servidor de Yafrid se ha desarrollado sobre una arquitectura en la que, de forma general, se pueden distinguir cuatro capas o niveles. Estas capas son, de menor a mayor nivel de abstracción, las siguientes:

- Capa de Recursos.
- Capa de Servicios.
- Capa Yafrid.
- Capa de Usuario.

Es en la tercera de estas capas donde el verdadero Servidor Yafrid reside. El resto de capas se consideran capas de soporte pero son igualmente indispensables para la implantación del sistema en un entorno real.

Capa de Recursos

Es la capa que tiene el nivel de abstracción más bajo y la que posee una relación más estrecha con el sistema operativo. La capa de recursos engloba los siguientes componentes:

- **Sistema de Base de Datos.** Formado por una base de datos en la que están contenidas las distintas tablas para el correcto funcionamiento del sistema. Existen numerosas tablas que se pueden agrupar atendiendo a su utilidad. Mientras que unas se usan para mantener datos a partir de los cuales se obtendrán estadísticas otras mantienen los datos que ayudan a la gestión de usuarios, grupos, proyecto, etc. Hay un tercer grupo que podríamos llamar internas o de operación que mantienen datos imprescindibles para el funcionamiento del sistema como por ejemplo los datos de conexión de los proveedores o las características de sus sistemas. Los componentes de capas superiores acceden a todos estos datosa través de un servidor de base de datos. La actual implementación de Yafrid utiliza MySQL.

- **Sistema de Archivos.** En ocasiones es necesario acceder al sistema de archivos desde capas superiores. Básicamente, se pueden distinguir dos tipos de directorios. Por un lado, hay directorios que se usan para almacenar las unidades de trabajo pertenecientes a un proyecto lanzado. Los proveedores accederán a este directorio vía SFTP para recibir los ficheros fuente necesarios para llevar a cabo el render. Estos directorios forman el llamado POOL de unidades de trabajo. La otra categoría la forman aquellos directorios que contienen la información sobre los usuarios y sus proyectos.

- **Sistema de Red.** El módulo dedicado a las comunicaciones que pertenece a la capa principal oculta la utilización de los recursos de red del ordenador por medio de un middleware (la implementación actual utiliza ICE).

Capa de Servicios

Básicamente, esta capa contiene los diferentes servidores que permiten a los módulos de capas superiores acceder a los recursos que pertenecen a la capa por debajo de ésta. En este nivel se pueden encontrar los siguiente servidores:

- **Servidor HTTP.** El módulo Yafrid-WEB se establece sobre este servidor. Como el módulo Yafrid-WEB se ha confeccionado usando páginas dinámicas escritas en un lenguaje de script orientado al desarrollo web (en la implementación actual se ha utilizado PHP), el servidor web debe tener soporte para este lenguaje. También es necesario soporte para la composición de gráficos y para acceso a base de datos.
- **Servidor de bases de datos.** Este servidor se usa por los diferentes módulos de Yafrid para acceder a datos imprescindibles para la operación del sistema.
- **Servidor SFTP.** Accedido por los distintos proveedores de servicio para obtener los ficheros fuente necesarios para llevar a cabo los trabajos de render. Una vez que el render haya finalizado, se usará el servidor SFTP para enviar de vuelta al servidor las imágenes resultantes.

Capa Yafrid

Constituye la capa principal del servidor y está compuesta básicamente por dos módulos diferentes que funcionan de forma independiente el uno del otro. Estos módulos se denominan Yafrid-WEB y Yafrid-CORE.

Yafrid-WEB

Constituye el módulo interactivo del servidor y ha sido desarrollado como un conjunto de páginas dinámicas escritas usando HTML y un lenguaje de script orientado a desarrollo web.

En términos de acceso al sistema, se han definido tres roles de usuario que determinan los privilegios de acceso de los usuarios a la intefaz web de Yafrid. Estos roles junto con las funcionalidades a las que dan acceso son:

1. **Cliente**. Tener este rol permite a un usuario el envío de trabajos de render al grid. Un cliente es también capaz de crear y gestionar grupos a los que otros clientes y proveedores se pueden suscribir. Cuando un proyecto se crea, se le puede asignar un grupo privado. En este caso, sólo los proveedores pertenecientes al mismo grupo pueden tomar parte en el render del proyecto. Además, se generan diversas estadísticas con la información relativa a los proyectos del usuario.

2. **Administrador**. Este usuario es imprescindible para la operación del sistema y tiene privilegios que permiten acceder a información y funcionalidades críticas que incluyen:
    - Acceso a la información de los usuarios del sistema (clientes y proveedores) y de los grupos de render existentes.
    - Creación de conjuntos de pruebas, un tipo especial de proyectos compuestos que están formados por múltiples proyectos individuales con ligeras diferencias entre unos y otros. Estos conjuntos de pruebas constituye una herramienta con la que estudiar el rendimiento del sistema y obtener conclusiones a partir de los resultados.
    - Gestión de la parte no interactiva del sistema (Yafrid-CORE).
3. **Proveeder**. El proveedor es el usuario que tiene instalado el software necesario para formar parte del grid. Los proveedores pueden acceder a su propia información y a las estadísticas de uso.

Los grupos de render mencionados tienen una especial importancia en un grid dedicado a tareas de render como es Yafrid. Con este sencillo mecanismo, los proyectos pueden ser creados dentro de un grupo teniendo la garantía de que los proveedores de ese grupo dedicarán sus esfuerzos al render de ese proyecto y no al de otros.

Yafrid-CORE

Esta es la parte no interactiva del servidor. Este módulo ha sido principalmente desarrollado usando Python aunque también existen algunos scripts en Bourne shell para tareas de gestión. Está compuesto por dos módulos:

1. **Distribuidor**. Constituye la parte activa del módulo. Implementa el algoritmo principal destinado a realizar una serie de tareas clave, algunas de las cuales se introducen a continuación.
    - **Generación de unidades de trabajo.** Consiste básicamente en lanzar los proyectos activos que existen en el sistema generando las unidades de trabajo que sean necesarias. Las características de esta división dependen de los parámetros que el usuario introdujo al activar el proyecto.

- **Asignación de las unidades de trabajo** a los proveedores. Esta es la tarea principal del proceso de planificación. El algoritmo tiene que tener en cuenta factores como el software que es necesario para renderizar la escena, el software instalado en los proveedores y la versión específica de éste o el grupo al que pertenecen proveedor y proyecto. Con toda esta información, y alguna más, el Distribuidor tiene que decidir qué unidad será enviada a qué proveedor.

- **Composición de resultados.** Con lso resultados generados por los diferentes proveedores, el distribuidor tiene que componer el resultado final. Este proceso no es en absoluto trivial, en especial, en los render de escenas estáticas. Debido al componente aleatorio de los métodos de render basados en técnicas de Monte Carlo (como el Pathtracing), cada uno de los fragmentos puede presentar pequeñas diferencias cuando se renderizan en distintas máquinas. Por esta razón, es necesario suavizar las uniones entre fragmentos contiguos usando una interpolación lineal. En la Figura 3 (a la izquierda) se pueden observar los problemas al unir los fragmentos sin usar este mecanismo de suavizado.

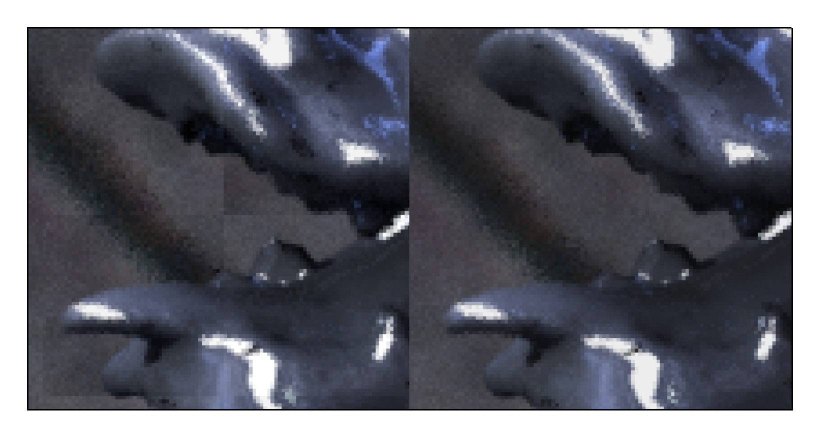


Figura 3. Izqda: *Artefactos sin interpolación entre fragmentos.* Drcha: *Usando interpolación lineal.*

- **Finalización de proyectos.** Los resultados parciales almacenados por el sistema durante el proceso son eliminados del mismo.

- **Control de timeout.** Cuando el tiempo máximo que está establecido que se puede esperar por una unidad de trabajo es sobrepasado, el sistema consulta el estado de la unidad. Se comprueba si la unidad enviada a un proveedor está siendo aún procesada o no. Si una de las unidades se pierde, ésta tiene que ser reactivada para ser enviada a otro proveedor en un futuro.

2. **Identificator**. Constituye la parte pasiva del módulo Yafrid-CORE cuya misión consiste en mantenerse a la espera de comunicaciones por parte de los proveedores. Este módulo es responsable del protocolo que se establece entre el servidor y los proveedores para garantizar el correcto funcionamiento del sistema. Se encarga de todas las comunicaciones con los proveedores excepto aquella en la que una unidad de trabajo es enviada a un proveedor, que es realizada por el distribuidor.

La primera vez que un proveedor contacta con el servidor de Yafrid, el Identificador genera un objeto, el controlador, y devuelve al proveedor un proxy que apunta a este objeto. Cada proveedor posee su propio controlador. Las siguientes interacciones entre el proveedor y el servidor se realizarán a través del controlador del proveedor. Algunas de estas interacciones representan operaciones como la identificación, la suscripción, la activación, la desconexión, el aviso de que una unidad ha sido terminada, etcétera.

Capa de Usuario

Esta capa no forma parte del sistema en un sentido estricto. Básicamente la constituyen los distintos tipos de usuarios que pueden acceder al sistema. Como ya se ha comentado, existen tres roles que caracterizan las tres categorías de usuarios de Yafrid:

1. Proveedor de servicio.
2. Cliente.
3. Administrador de Yafrid.

Estos usuarios pueden acceder a toda la información concerniente a su relación con el sistema ofrecida por el módulo Yafrid-WEB utilizando cualquier navegador (Figura 4).

Figura 4. *Interfaz de usuario Web – Yafrid.*

Esta caracterización de los distintos usuarios da lugar a algo similar a una sociedad en la que cada usuario tiene un papel determinado en la misma. En concreto, existen unos usuarios que proporcionan servicio a otros mientras que unos terceros gestionan el sistema.

Las características de cada uno de estos roles y de su interacción con el sistema serán abordadas en próximas secciones.

Proveedor

El Proveedor es el software que deben instalar en sus sistemas los usuarios que deseen prestar sus ciclos de CPU para que se usen en tareas de render. Este software puede funcionar tanto en modo de línea de comandos como por medio de una interfaz gráfica. El primer paso que un proveedor tiene que dar para formar parte del grid es conectarse al mismo. Una vez conectado y activado, el proveedor se pone a la espera de que el servidor le envíe una unidad de trabajo para ser procesada. Cuando finaliza el render, el proveedor envía los resultados de vuelta al servidor via SFTP e informa a su controlador de que el trabajo está terminado.

Resultados

En diversas etapas del desarrollo del sistema se han realizado pruebas con los distintos prototipos que se iban obteniendo. Mientras que en las primeras fases se realizaban pruebas sencillas en una sola máquina, cuando el sistema estuvo mínimamente maduro se comenzaron a realizar pruebas en un entorno distribuido y controlado. En esta sección se muestran algunos de los resultados obtenidos en estas pruebas así como los resultado esperados según los análisis previos.

Resultados analíticos

Para el siguiente estudio se suponen conexiones simétricas entre Servidor y Proveedores y capacidades de cálculo similares de los distintos Proveedores.

Como vemos en la Ecuación 1, al utilizar Yafrid para renderizar una escena se añaden dos factores al tiempo de render, T_{trans} y T_o, que no aparecen en un sistema no distribuido y con un solo ordenador.

El T_{trans} es el tiempo de transmisión y depende en parte de la velocidad de la red que une Servidor y Proveedores. T_o es un tiempo que el sistema utiliza en operaciones tales como la unión de las unidades de trabajo y que se puede acotar en funcion del tamaño del proyecto.

$$T_{total} = T_{render} + T_{trans} + T_0 \tag{1}$$

Tanto en Yafrid como en un sistema centralizado, el tiempo de render responde a la Ecuación 2 donde *fps* corresponde a los frames por segundo de la animación, t_{al} es la duración de la animación y t_{rf} es el tiempo medio en renderizar un frame.

$$T_{render} = \begin{cases} fps \times t_{al} \times t_{rf} & \text{En el caso de Animaciones} \\ t_{rf} & \text{En el caso de Frames} \end{cases} \tag{2}$$

El tiempo de transmisión se calcula mediante la Ecuación 3 en la que n_r es el número de Proveedores (o nodos de render), S_{pet} es el tamaño en bytes del fichero fuente que el Servidor envía a los Proveedores, S_{res} es el tamaño en bytes del fichero resultado que cada Proveedor envía al Servidor, W_i es el número de unidades de trabajo realizadas por el Proveedor i y $Vtrans_i$ es la velocidad de transmisión (ancho de banda) entre Servidor y Proveedores en bytes/s. El número de Proveedores es lo que se denomina tamaño del grid.

$$T_{trans} = \frac{\sum_{i=0}^{n_r} (S_{pet} + (S_{res} \times W_i))}{Vtrans_i} \tag{3}$$

Así, en un sistema con un sólo ordenador no aparecen los tiempos extra T_{trans} y T_o que sí aparecen en un sistema como Yafrid. Sin embargo, el proceso se desarrolla en paralelo en distintas máquinas con lo que, en un caso ideal, el tiempo total se divide equitativamente entre los distintos proveedores del sistema (Ecuación 4).

$$T = \frac{T_{total}}{n_r} \tag{4}$$

En el caso de que las redes que conectan los componentes del grid sean de alta velocidad, el término T_{trans} se puede despreciar, con lo que la ganancia sería del orden del número de proveedores n_r (n_r sería el valor máximo de la ganancia).

En futuras mejoras del sistema se pretende optimizar la generación de unidades de trabajo en proyectos de render estático de forma que éstas se rendericen con distintas calidades. Esto permitiría obtener ganancias por encima de n_r ya que la suma de los tiempos en renderizar cada uno de los fragmentos sería menor que el tiempo en renderizar la imagen completa debido a que algunos de los fragmentos se renderizan con una calidad menor que la inicial.

Resultados empíricos

Con el fin de sacar conclusiones sobre el comportamiento del sistema en un entorno real, se ejecutaron varios conjuntos de pruebas (test sets). Estos conjuntos de pruebas se pueden generar desde el interfaz web con permisos de administrador y consisten en una serie de proyectos de prueba (tests). Cada uno de los conjuntos de pruebas posee sus propios parámetros de calidad (como son por ejemplo el número de muestras de luz o el número de rayos por pixel) y su propio método de render. Dentro de un mismo conjunto de pruebas, los distintos proyectos de prueba se diferencian en el valor de los parámetros de calidad y en el tamaño de las unidades de trabajo en las que se dividirá el frame.

Para hacer que los datos que se obtuvieran pudieran ser comparables, se optó por configurar un grid homogéneo en el que todos los proveedores tuvieran similares características. Así, los proyectos de prueba mencionados se lanzaron en Yafrid mientras el grid tenía un tamaño de 16 proveedores idénticos en capacidades (procesador Pentium IV a 2.80 GHz, 256Mb de RAM y el mismo sistema operativo). Este tamaño de grid se mantuvo constante durante toda la prueba.

Para realizar las pruebas se seleccionó una imagen bastante compleja. La escena contiene más de 100.000 caras, 4 niveles de recursión en el algoritmo de trazado de rayos en superficies especulares (el dragón) y 6 niveles en superficies transparentes (el cristal de las copas). Además, se lanzaron 200.000 fotones para construir el mapa de fotones.

Para generar la imagen se utilizaron dos métodos de render:

- Pathtracing con Mapeado de Fotones e Irradiance Cache (Figura 5 - Izquierda).
- Trazado de Rayos clásico con una implementación de la técnica Ambient Occlusion (Figura 5 - Derecha).

Antes de presentar los resultados obtenidos con Yafrid (con un tamaño de grid de 16 ordenadores), en la Tabla 1 se muestran los tiempos de render de cada conjunto de pruebas obtenidos usando un solo proveedor (render local).





Figura 5. Izqda: *Pathtracing* Drcha: *Ambient Occlusion.*

Método	*Calidad*	*Tiempo (hh:mm:ss)*
Pathtracing	128 Muestras	02:56:56
Pathtracing	512 Muestras	07:35:28
Pathtracing	1024 Muestras	14:23:07
Pathtracing	2048 Muestras	23:16:12
RT Clásico	1 Rayo/Píxel	00:17:50
RT Clásico	8 Rayos/Píxel	00:20:39
RT Clásico	16 Rayos/Píxel	00:26:47

Tabla 1. *Resultados obtenidos al renderizar en una sola máquina.*

Como se puede observar en la Figura 6, el tiempo de render en el mejor de los casos es más de ocho veces mejor utilizando Yafrid. En el peor de los casos es tan sólo dos veces mejor.

Del mismo modo, utilizando el algoritmo de trazado de rayos de Whitted (Figura 6), el tiempo de render usando Yafrid es, en el mejor de los casos, siete veces menor y tan sólo tres veces menor en el peor de los casos.

También se lanzó una sencilla animación de prueba consistente en un movimiento de cámara en torno a los objetos principales de la escena usada en las pruebas anteriores. La duración de la animación se estableció en 50 fotogramas y se utilizó el algoritmo de trazado de rayos clásico.

Tal y como se muestra en la Tabla 2 en los proyectos de animación se consiguen ganancias más importantes que en el caso del render estático. La ganancia está en este caso más cercana al límite teórico que suponía el número de Proveedores. El tiempo obtenido con Yafrid es alrededor de catorce veces menor.

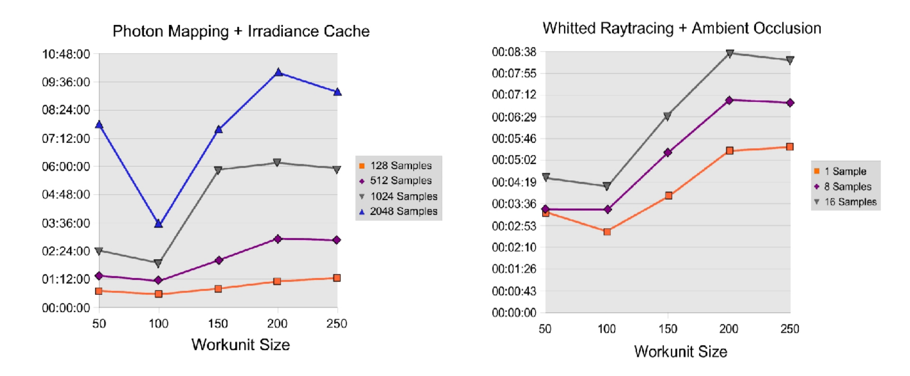


Figura 6. Izqda: *Resultados empleando Pathtracing*.
Drcha: *Resultados empleando Raytracing Clásico.*

Calidad	*Tiempo Yafrid*	*Tiempo 1 PC*
1 Rayo/Píxel	01:03:50	14:20:55
8 Rayos/Píxel	01:15:12	16:33:10
16 Rayos/Píxel	01:22:36	20:12:02

Tabla 2. *Resultados obtenidos al renderizar una animación.*

Análisis de los resultados

Con la observación de los resultados empíricos obtenidos en el apartado anterior, se pone de manifiesto la importancia que tiene la elección del tamaño de la unidad de trabajo en los proyectos de render estáticos. Para cada escena hay un tamaño óptimo de unidad de trabajo que permite obtener el mejor rendimiento posible.

En esta escena en concreto hay algunas unidades de trabajo que tardan mucho más que el resto (como las caústicas de las copas de cristal). Esto hace que el proyecto se mantenga en proceso gran parte del tiempo con únicamente algunos fragmentos por terminar. Esto enmascara la ganancia obtenida perjudicando el tiempo que se ha dedicado al proyecto.

En el método usado por Yafray para realizar el render (Pathtracing), los fragmentos a calcular no son totalmente independientes unos de otros. Para obtener las unidades de trabajo, cada Proveedor tiene que calcular la caché de irradiancia completa cada vez. La mejora aumentaría mucho si un Proveedor calculara la caché una vez y se la enviara a los demás Proveedores (mecanismo aún no implementado en Yafrid).

La ganancia es mayor en el método de trazado de rayos de Whitted debido a que no se hace uso de la caché de irradiancia.

Además existe otro factor por el que la ganancia es menor que la teórica. A la hora de dividir una imagen en fragmentos se añade un área extra, la banda de interpolación, que servirá para suavizar las uniones entre unos fragmentos y otros. Usar una banda de interpolación menor -en los ejemplos tiene una anchura de 10 píxeles- mejoraría la ganancia.

Con respecto a las animaciones, la situación es bastante distinta. La ganancia obtenida experimentalmente está más cerca de ser proporcional al tamaño del grid que en el caso del render de un solo frame, es decir, está más cerca de la ganancia teórica. Una de las causas es que, habitualmente, el tiempo de render es más homogéneo entre un frame y el siguiente que entre fragmentos de un mismo frame. Además, el tiempo invertido en unir los frames de una animación es menor que el invertido en unir e interpolar los fragmentos de un frame. Otra razón importante es que no hay que renderizar un área extra ya que no existe banda de interpolación.

Conclusiones sobre el rendimiento del sistema

De acuerdo con los resultados empíricos examinados con anterioridad, el sistema proporciona claramente un mejor rendimiento que en el caso del render en un sistema con un sólo procesador. Esto es así incluso cuando la elección del tamaño de la unidad de trabajo es la peor de las posibles.

Yafrid ofrece una significativamente mejor **productividad** (número de entidades renderizadas, frames o fragmentos de un frame, por unidad de tiempo) que un único ordenador porque las distintas unidades de trabajo se generan paralelamente en distintos proveedores. Esta productividad depende en gran medida del tamaño de la unidad de trabajo y del tamaño del grid.

Por otro lado, la **latencia** de cada una de las unidades (el tiempo que tardan en ser generadas) se incrementa porque hay que tener en cuenta factores que no existen en el procesamiento no distribuido. En sistemas distribuidos de cualquier tipo, además del tiempo dedicado a renderizar la escena, el tiempo de transmisión es también importante. Este tiempo de transferencia depende de la cantidad de datos a transferir, de la velocidad de la conexión y del estado de la red.

El aumento de la latencia no es importante si obtenemos una mayor productividad, que es lo que en realidad va a percibir el usuario final del servicio.

Por regla general, cuanto mayor sea el número de fragmentos en los que se divide una escena, mayor paralelismo se obtendrá y menos tiempo se necesitará para renderizar la imagen completa. Si analizamos el tema con más detalle, esto no es tan simple. Si una escena se divide en fragmentos demasiado pequeños es posible que el tiempo dedicado a la transmisión y al procesamiento de los resultados sea mayor que el tiempo dedicado al render. En este caso, los resultados obtenidos por un sistema distribuido de cualquier tipo serán peores que los obtenidos en una sola máquina. Una elección incorrecta del tamaño de la unidad de trabajo puede perjudicar el rendimiento del grid.

Estudio de Métodos de Render

Carlos González Morcillo
Carlos.Gonzalez@uclm.es ::



A un alto nivel de abstracción podemos ver el proceso de *Render* como el encargado de convertir una descripción de una escena tridimensional en una imagen. Prácticamente cualquier disciplina de aplicación de los gráficos 3D (videojuegos, cine, visualización científica, marketing, etc...) necesitan obtener resultados visibles mediante un proceso de *Rendering*.

La naturaleza interdisciplinar del proceso de *Render* se puede comprobar en que multitud de disciplinas científicas hacen uso de algortimos y técnicas relacionadas con este campo, como física, astrofísica, biología, astronomía, psicología y estudio de la percepción, matemáticas aplicadas...

En los primeros años de estudio de este campo la investigación se centró en cómo resolver problemas relacionados con la detección de superficies visibles, sobreado básico, etc... Según se encontraban soluciones a estos problemas, se continuó el estudio en algoritmos de síntesis más realistas que simularan el comportamiento de la luz con la mayor fidelidad posible. En la actualidad, el grado de realismo alcanzado es muy alto, siendo unas de las principales áreas de investigación en este campo las técnicas de optimización en tiempos de ejecución.

Síntesis de Imagen Fotorrealista

Podemos definir el proceso de **síntesis de imagen fotorrealista** como aquel que persigue la creación de imágenes sintéticas que no se puedan distinguir de aquellas captadas en el mundo real. En este campo de los gráficos por computador ha ocurrido una rápida evolución. Desde el algoritmo de Scan-Line, propuesto por Bouknight [BOU70] que no realizaba ninguna simulación física de la luz, hasta los inicios del fotorrealismo, en el artículo de Whitted [WHI80] con la propuesta del método de trazado de rayos y posteriormente la introducción de la Radiosidad [GOR84] hicieron uso de técnicas habituales en el campo de la óptica y el cálculo de transferencia de calor.

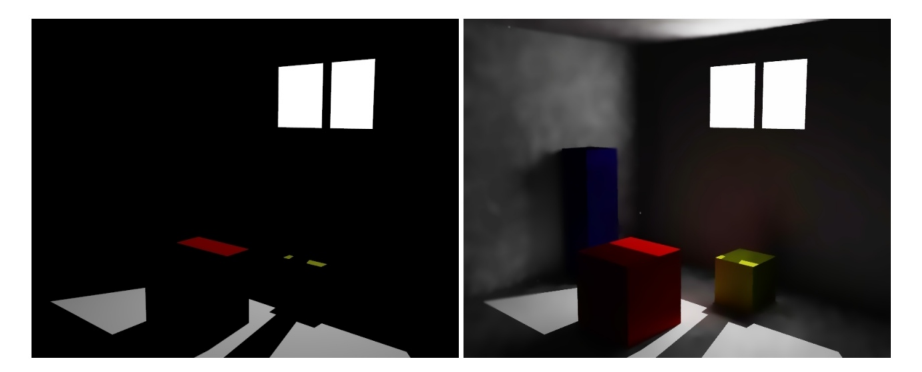


Figura 1. Izqda: *Escena renderizada aplicando únicamente iluminación directa*
Drcha: *Escena renderizada con iluminación global empleando mapeado de fotones; se observa la caja de color azul pegada a la pared que no recibe iluminación directa.*

En la búsqueda de la generación de imagen fotorrealista un punto clave es la simulación de la luz. La simulación siguiendo las leyes físicas de toda la luz dispersa en un modelo 3D sintético se denomina **iluminación global** (*global illumination*). El objetivo de las técnicas de iluminación global es simular todas las reflexiones de la luz y así obtener en cada punto del modelo el valor de intensidad de luz que vendrá determinado por la interacción de la luz con el resto de elementos de la escena. De esta forma, el objetivo del algoritmo de iluminación global es calcular la interacción de todas las fuentes de luz con los elementos que forman la escena. Estas técnicas de iluminación global son necesarias para calcular la iluminación indirecta; sin esta iluminación el modelo parece artificial y sintético (ver Figura 1). Existen varias técnicas de iluminación global, pero la mayoría están basadas en el **Trazado de Rayos** (o Muestreo de Puntos) o en **Radiosidad** (o Elementos Finitos) o en ambas a la vez (técnicas híbridas). Cada uno de estos métodos tiene sus ventajas e inconvenientes.

Técnicas de Trazado de Rayos

Mediante la técnica de trazados de rayos se trazan un alto número de rayos de luz a través del modelo 3d que se pretende renderizar. En este modelo, introducido en 1980 por Whitted [WHI80], se trazan rayos desde el observador hacia las fuentes de luz. Algunas limitaciones importantes de este tipo de trazado de rayos básico es que no puede representar profuncidad de campo, caústicas o iluminación indirecta. Para simular estos efectos, es necesario extender el trazado de rayos empleando **métodos de Monte Carlo** [COO86], que lancen rayos adicionales para simular todos los caminos posibles de la luz. Los métodos de Monte Carlo tienen un problema importante debido a su **varianza**, que se percibe como ruido en las imágenes generadas.

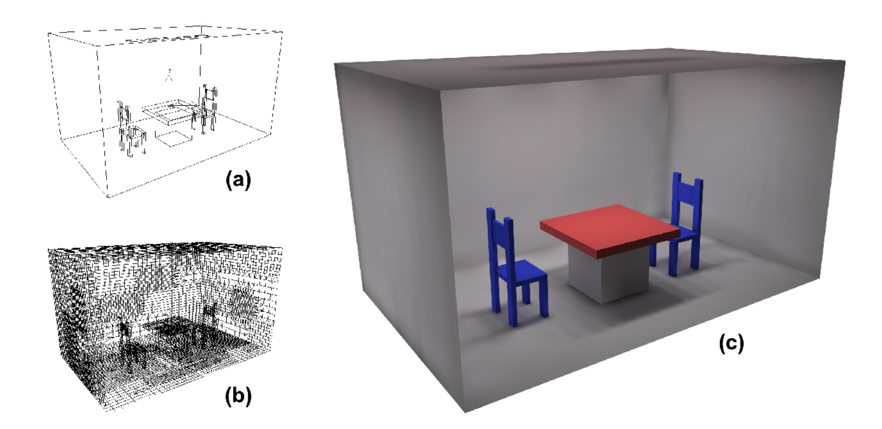


Figura 2: *Ejemplo de uso de radiosidad. a) Malla original; en el techo de la habitación se observan las caras emisoras de luz. b) Malla de radiosidad resultado. c) Render de la habitación con los valores de radiosidad calculados en (b).*

La eliminación de este ruido requiere aumentar el número de muestras. Se han realizado numerosos estudios sobre cómo distribuir los rayos de forma que el ruido sea menor. Las alternativas que mejores resultados han dado hasta la fecha son los métodos guiados de trazado de rayos, cuyas propiedades de convergencia son mucho mejores (como por ejemplo las técnicas de irradiance caché [WAR92], que almacenan y reutilizan la iluminación indirecta interpolando los valores desconocidos).

En este tipo de técnicas la geometría se trata como una "caja negra". Los rayos se trazan en la escena y obtienen un valor de iluminación. Esta es una importante ventaja en escenas con una geometría compleja, ya que las técnicas de Muestreo de Puntos sólo tienen que gestionar la complejidad de la iluminación, y no de la geometría. Sin embargo, esta independencia de la geometría tiene inconvenientes, como la necesidad de un alto número de muestras para unos resultados aceptables.

Técnicas de Radiosidad

En este tipo de técnicas se calcula el **intercambio de luz entre superficies**. Esto se consigue subdividiendo el modelo en pequeñas unidades denominadas "parches", que serán la base de la distribución de luz final. Inicialmente los modelos de radiosidad calculaban las interacciones de luz entre superficies difusas (aquellas que reflejan la luz igual en todas las direcciones) [COH85] [GOR84], aunque existen modelos más avanzados que tienen en cuenta modelos de reflexión más complejos.

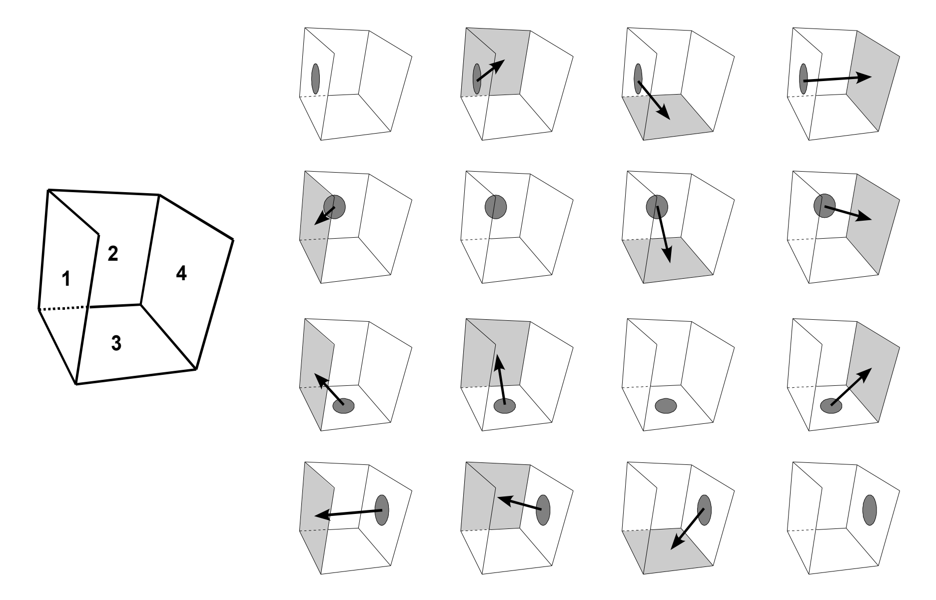


Figura 3: *Esquema de la matriz de radiosidad. En cada posición de la matriz se calcula el factor de forma. La diagonal principal de la matriz no es calculada.*

El modelo básico de radiosidad calcula una **solución independiente del punto de vista**. Sin embargo, el cálculo de la solución es muy costoso en tiempo y en espacio de almacenamiento. No obstante, cuando la iluminación ha sido calculada, puede utilizarse para renderizar la escena desde diferentes ángulos, lo que hace que este tipo de soluciones se utilicen en visitas interactivas y videojuegos en primera persona actuales.

En el modelo de radiosidad, cada superficie tiene asociados dos valores: la intensidad luminosa que recibe, y la cantidad de energía que emite (energía Radiante). En este algoritmo se calcula la interacción de energía desde cada superficie hacia el resto. Si tenemos n superficies, la complejidad del algoritmo será $O(n^2)$. El valor matemático que calcula la relación geométrica entre superficies se denomina **Factor de Forma**. Será necesario calcular n^2 factores de forma ya que no cumple la propiedad conmutativa debido a que se tiene en cuenta la relación de área entre superficies. La matriz que contiene los factores de forma relacionando todas las superficies se denomina **Matriz de Radiosidad**. Cada elemento de esta matriz contiene un factor de forma para la interacción desde la superficie indexada por la columna hacia la superficie indexada por la fila (ver figura 3).

En general, el cálculo de la radiosidad es eficiente para el cálculo de distribuciones de luz en modelos simples con materiales difusos, pero resulta muy costoso para modelos complejos (debido a que se calculan los

valores de energía para cada parche del modelo) o con materiales no difusos. Además, la solución del algoritmo se muestra como una nueva malla poligonal que tiende a desenfocar los límites de las sombras. Esta malla poligonal puede ser inmanejable en complejidad si la representación original no se realiza con cuidado o si el modelo es muy grande, por lo que este tipo de técnicas no se utilizan en la práctica con modelos complejos.

Técnicas Mixtas

La radiosidad es buena para simular las reflexiones difusas, mientras que el trazado de rayos da mejores resultados en reflexión especular. De este modo podemos utilizar trazado de rayos para el cálculo de las reflexiones especulares y radiosidad para superficies con reflexión difusa. Existen varias aproximaciones en las que se utiliza radiosidad para el cáculo de iluminación indirecta, ya que las técnicas de Monte Carlo requieren un alto número de muestras para que no se perciba el ruido. El uso de algoritmos basados en radiosidad tienen un importante inconveniente y es que limitan la complejidad de los modelos que pueden ser renderizados. Una aproximación a estos problemas se basa en simplificar la malla origen para utilizarla únicamente en el cálculo de la iluminación indirecta cuyos cambios suelen ser suaves y no requieren un alto nivel de detalle en la superficie.

El **mapeado de fotones** [JEN01] se basa en una aproximación difentente, ya que separa la información de la iluminación de la geometría, almacenándola en una estructura de datos diferente (el *mapa de fotones*). Este mapa se construye trazando fotones desde las fuentes de luz, y guarda información sobre los impactos de los fotones, que se utilizará para renderizar la escena de una forma eficiente, tal y como se utilizaba la información sobre la iluminación en la radiosidad. Este desacople de la geometría permite simplificar la representación y utilizarla en modelos de cualquier nivel de complejidad.

Estudio de Métodos de Renderizado

A continuación vamos a describir algunos métodos de renderizado ampliamente utilizados en la actualidad. Nos centraremos en los métodos de renderizado Píxel a Píxel (frente a las alternativas basadas en primitivas, que concluyen en una etapa de rasterización).

Podemos considerar que todas las alternativas de renderizado pretenden dar solución a la ecuación de renderizado, propuesta en 1986 por Kajiya [KAJ86]:

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w})\, L_i(x,\vec{w}')(\vec{w}'\cdot\vec{n})\, d\vec{w}'$$

Que puede leerse como: "dada una posición x y dirección $\vec{w}$ determinada, el valor de iluminación saliente L_o es el resultado de sumar la iluminación emitida L_e y la luz reflejada. La luz reflejada (el segundo término de la suma de la ecuación anterior) viene dado por la suma de la luz entrante L_i desde todas las direcciones, multiplicado por la reflexión de la superficie y el ángulo de incidencia.

Para expresar la interacción de la luz en cada superficie se utiliza la función de distribución de reflectancia bidireccional (BRDF), que correspondería con el término f_r de la expresión anterior y es particular de cada superficie.

En la medida de lo posible trataremos de responder a las siguientes preguntas sobre cada método: ¿En qué consiste?, ¿Quién lo propuso?, ¿Cuándo se descubrió?, ¿Por qué surgió?, ¿Cual es la descripción de su algoritmo?, así como una enumeración de ventajas y desventajas frente a otros métodos de render.

Scanline

Es un método propuesto en 1970 por Bouknight [BOU70], como algoritmo para la representación punto a punto de una imagen. Cuando se calculan los píxeles que forman cada línea de la imagen, se continúa con la siguiente.

Descripción del Algoritmo

```
para cada píxel de la imagen
linea := trazar una línea desde la cámara al píxel
color = scanline(linea)

scanline(linea)
punto := encontrar el punto de intersección más cercano
color := color de fondo
para cada fuente de luz
  color := color + iluminacion directa
devolver(color)
```

Ventajas

Es un método muy rápido, que permite simular cualquier tipo de sombreado, funciona correctamente con texturas.

Desventajas

No permite simular de forma realista reflexiones y refracciones de la luz. En el caso de las reflexiones pueden simularse mediante mapas de entorno (que dan una aproximación rápida, pero no físicamente correcta). Las refracciones no son correctas, aunque en cuerpos planos pueden ser medianamente realistas simulándolas con transparencia a nivel de polígono.

Raytracing

El método de trazado de rayos es una propuesta elegante y recursiva que permite calcular de forma sencilla superficies con reflejos y transparencia. Fue propuesto en 1980 por Whitted [WHI80].

La idea base en Raytracing es la de trazar rayos desde el observador a las fuentes de luz. En realidad, son las fuentes de luz las que emiten fotones que rebotan en la escena y llegan a los ojos del observador. Sin embargo, sólo una pequeñísima fracción de los fotones llegan a este destino, por lo que el cálculo en esta forma directa resulta demasiado costosa. Los rayos que se emiten a la escena son evaluados respecto de su visibilidad trazando nuevos rayos desde los puntos de intersección (rayos de sombra).

Descripción del Algoritmo

```
para cada píxel de la imagen
rayo := trazar un rayo desde la cámara al píxel
color = raytracing(rayo)

raytracing(rayo)
punto := encontrar el punto de intersección más cercano
color := color de fondo
para cada fuente de luz
  rayo_sombra  :=  trazar  un  rayo  desde  (punto)  hasta  la
fuente de luz
  si el rayo no choca con ningún objeto
    color := color + iluminacion directa
    si el material tiene propiedad de reflexión
      color := color + raytracing(rayo_reflejado)
    si el material tiene propiedad de refracción
      color := color + raytracing(rayo_transmitido)
  sino
    color := negro
  devolver (color)
```

Existen métodos de aceleración del algoritmo de raytracing, como el uso de árboles octales, grids jerárquicos, árboles BSP, etc... Además, se pueden calcular los valores de píxeles cercanos y si la variación es pequeña interpolar los valores (irradiance cache).

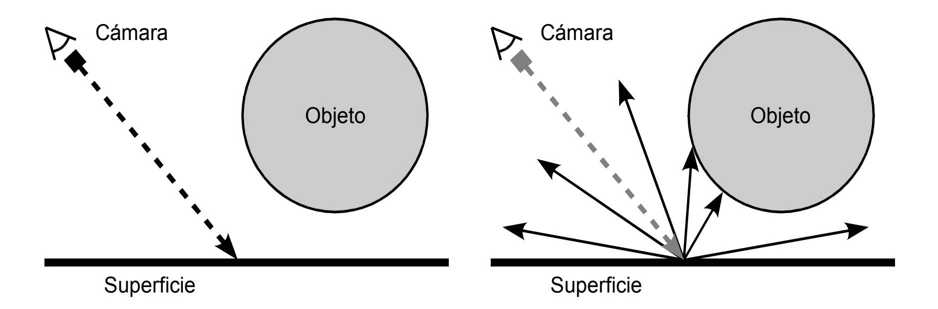


Figura 4. *Pasos para el cálculo de Ambient Occlusion. La figura de la izquierda representa el primer paso del método, donde se calcula el punto P. El valor de ocultación W(P) en la figura derecha será proporcional al número de rayos que alcanzan el "cielo", teniendo en cuenta la distancia máxima entre P y el primer objeto de intersección. En el caso de la figura de la derecha será de 1/3 de los rayos.*

Ventajas

Permite simular reflexiones de espejo y materiales transparentes (refracción de la luz).

Desventajas

No es un método de iluminación global. No permite la simulación de iluminación indirecta ni sombras difusas. Sólo funciona perfectamente en superficies con brillo especular donde la luz entrante tenga una única dirección de reflexión.

No permite simular profundidad de campo ni motion blur (aunque pueden simularse en postproducción).

Ambient Occlusion

El empleo del término de luz ambiente viene aplicándose desde el inicio de las técnicas de síntesis de imagen 3D. La técnica de Ambient Occlusion es un caso particular del uso de pruebas de oclusión en entornos con iluminación local para determinar los efectos difusos de iluminación. Fueron introducidos inicialmente por Zhurov en 1998 como alternativa a las técnicas de radiosidad para aplicaciones interactivas (videojuegos), por su bajo coste computacional [ZHU98]. Podemos definir la ocultación de un punto de una superficie como:

$$W(P)=\frac{1}{\pi}\int_{\omega\in\Omega}\rho(d(P,\omega))\cos\theta\, d\omega$$

Obteniendo un vador de ocultación *W(P)* entre 0 y 1, siendo *d(P,ω)* la distancia entre *P* y la primera intersección con algún objeto en la dirección de *ω*, *ρ(d(P,ω))* es una función con valores entre 0 y 1 que nos indica la magnitud de iluminación ambiental que viene en la dirección de *ω*, y *θ* es el ángulo formado entre la normal en P y la dirección de *ω*.

Estas técnicas de ocultación (*obscurances*) se desacoplan totalmente de la fase de iluminación local, teniendo lugar en una segunda fase de iluminación secundaria difusa. Se emplea un valor de distancia para limitar la ocultación únicamente a polígonos cercanos a la zona a sombrear mediante una función. Si la función toma como valor 0 de ocultación aquellos puntos que no superan un umbral y un valor de 1 si están por encima del umbral, la técnica de ocultación se denomina Ambient Occlusion. Existen multitud de funciones exponenciales que se utilizan para lograr estos efectos.

Ventajas

Es al menos un orden de magnitud más rápido que el PathTracing. En muchos casos la simulación obtenida es suficientemente buena y con menos ruido.

Desventajas

No es un método de iluminación global, aunque ofrece aproximaciones bastante buenas. No permite la simulación de caústicas, y puede resultar costoso si aumentamos el número de muestras (para eliminar ruido). El tiempo de render aumenta muy rápido con el número de samples de Antialiasing.

Radiosidad

En este tipo de técnicas se calcula el intercambio de luz entre superficies. Inicialmente fue propuesto por Goral en 1984 [GOR84]. El cálculo del intercambio de luz entre superficies se hace mediante el factor de forma, que se calcula mediante la siguiente expresión:

$$F_{i,j}=\frac{1}{A_i}\int_{A_i}\int_{A_j}\frac{\cos\theta_i\cos\theta_j}{\pi r^2}H_{ij}\,dA_j\,dA_i$$

Siendo $F_{i,j}$ el factor de forma de la superficie *i* a la superficie *j*, $\cos\theta_i \cos\theta_j$ es el ángulo entre las normales de los planos de cada parche (esto nos asegura que una superficie que esté en frente de otra que emite luz recibirá más luz que una tercera superficie que esté inclinada a ésta). πr^2 nos mide la distancia entre los parches y H_{ij} es el factor de visibilidad, entre 0 y 1, que indica cuánta superficie de *j* ve la superficie *i* debido a la oclusión de otras superficies. Finalmente dA_x es el área de la superficie *x*.

Ventajas

Ofrecen buenos resultados con superficies difusas. Al ofrecer una solución independiente del punto de vista, pueden emplearse para renderizar animaciones incluso de forma interactiva.

Desventajas

Limitan la complejidad de la geometría con la que podemos trabajar. Tiende a suavizar las zonas de penumbra, lo cual no siempre es deseable. El tiempo de cálculo de la matriz de radiosidad puede ser muy alto, incluso trabajando con técnicas incrementales. No funcionan bien con sombreado especular.

Pathtracing

Es una extensión del algoritmo de Raytracing clásico que permite calcular una solución de iluminación global completa. Fue formulado por Kajiya en 1986 como solución a la ecuación de renderizado (en el mismo artículo donde era propuesta) [KAJ86], basándose en las ideas de Cook en su artículo de 1984 [COO84].

El Pathtracing se basa en la idea de calcular *todos* los posibles caminos de la luz en sus rebotes en superficies que no tengan reflejo especular. Estas superficies requieren la evaluación de integrales complejas. Estas integrales se solucionan trazando rayos "aleatorios" dentro del dominio de la integral para estimar su valor (empleando integración de *Monte Carlo*). Realizando la media de un alto número de muestras por cada píxel podemos obtener una estimación del valor de la integral para ese píxel.

Un aspecto importante del Pathtracing es que sólo utiliza un rayo reflejado para estimar la iluminación indirecta. Es decir, cuando un rayo choca sobre una superficie difusa, sólo se realizará una llamada con un nuevo rayo de cálculo de la iluminación indirecta, cuya dirección será aleatoria dentro del dominio de definición de la superficie. Se podrían tener en cuenta múltiples rayos, pero al ser un método recursivo, el número de rayos crecería de forma exponencial si estos nuevos rayos chocaran con superficies difusas...

Descripción del Algoritmo

```
para cada píxel de la imagen
color := 0
para cada muestra del píxel
  rayo := trazar un rayo desde la cámara a una posición
           aleatoria del píxel
  color := color + pathtracing(rayo)
color_pixel := color / numero_muestras

pathtracing(rayo)
punto := encontrar el punto de intersección más cercano
color := color de fondo
para cada fuente de luz
  comprobar visibilidad de una posición aleatoria de la
     fuente de luz
  si es visible
    color := color + iluminacion directa
    color := color + pathtracing(un rayo reflejado
aleatorio)
  sino
    color := negro
  devolver (color)
```

Ventajas

Si la iluminación varía poco (por ejemplo, una escena exterior donde el cielo es totalmente visible), ofrece resultados sin ruido con pocas muestras.

Desventajas

Para que no se perciba ruido en la imagen final hay que aumentar el número de muestras por píxel, sobre todo en escenas donde hay variaciones muy fuertes en las fuentes de luz (fuentes pequeñas y muy potentes, o imágenes HDRI con zonas de iluminación muy concentradas).

Debido a las propiedades de integración de Monte Carlo, el error disminuye proporcionalmente a $1/\sqrt{N}$, siendo N el número de muestras. Esto significa que para eliminar la mitad de ruido tenemos que utilizar cuatro veces más muestras.

Irradiance Cache

Es un método que evita el cálculo de la iluminación global en cada punto del modelo. Se seleccionan algunos píxeles y se interpolan los valores intermedios. Esto hace que se suavice el ruido (aunque se obtiene ruido de baja frecuencia). Funciona bien con imágenes estáticas, pero se percibe ruido en animaciones.

Photon Mapping

Es un método que desacopla la representación de la iluminación de la geometría. Se realiza en dos pasos, primero se construye la estructura del mapa de fotones (trazado de fotones), desde las fuentes de luz al modelo. En una segunda estapa de render se utiliza la información del mapa de fotones para realizar el renderizado de manera más eficiente.

Este método fue propuesto por Jensen en 1996 [JEN96]. En una primera etapa un elevado número de fotones son lanzados desde las fuentes de luz, dividiendo la energía de las fuentes de luz entre los fotones emitidos (cada fotón transporta una fracción de la energía de la fuente de luz).

Cuando se emite un fotón, éste es trazado a través de la escena de igual forma que se lanzan los rayos en raytracing, excepto por el hecho de que los fotones propagan Flux en vez de radiancia. Cuando un fotón choca con un objeto puede ser reflejado, transmitido o absorbido (según las propiedades del material y un factor aleatorio dentro del dominio del material).

Los fotones son almacenados cuando impactan sobre superficies no especulares (para reflejos se utiliza raytracing clásico). Este almacenamiento se realiza sobre una estructura de datos que está desacoplada de la geometría del modelo, y se utilizará en la etapa de render para obtener información sobre qué impactos de fotones están más cerca del punto del que queremos calcular su valor de iluminación. Para que la búsqueda sea rápida se suelen emplear kd-trees (debido a que la distribución de fotones no es uniforme), cuyo tiempo medio de búsqueda con n fotones es de *O(log n)*, y *O(n)* en el peor de los casos.

Pathtracing Bidireccional

Fue introducido por Lafortune y Willems en 1993 [LAF93]. Se planteó como una extensión del Pathtracing (en realidad el Pathtracing puede considerarse como un caso particular de esta técnica) donde además se calculan los caminos desde las fuentes de luz. Esto aprovecha la idea de que hay ciertos caminos que son más fáciles de trazar desde las fuentes de luz, como por ejemplo el caso de las caústicas. La probabilidad de que un rayo reflejado en una superficie atraviese un objeto transparente y llegue a la fuente de luz es muy pequeña comparada con el camino inverso (desde la fuente de luz a la superficie).

En el Pathtracing bidireccional se trazan dos caminos en la escena, uno que comienza en cada fuente de luz y otro en el punto de vista. A continuación se combina la información de los dos caminos conectando los vértices de cada camino, y calculando la contribución individual de cada vértice del camino de la luz al camino trazado desde la cámara.

Finalmente la imagen se calcula como en pathtracing, realizando un número de muestras por píxel.

Ventajas

Se utiliza un número menor de muestras por píxel que en Pathtracing.

Se comporta mucho mejor con los casos extremos donde Pathtracing sufre de mucho ruido (como fuentes de iluminación pequeñas con mucha energía, por ejemplo, una pequeña ventana por donde entra la luz a una habitación). Para escenas de exteriores o con una iluminación más homogénea, es mejor utilizar pathtracing.

Las caústicas se calculan perfectamente (en pathtracing habría que recurrir a su combinación con Photon Mapping).

Desventajas

Aunque el número de muestras es menor, el tiempo de cálculo de la escena suele ser mayor, debido a que el coste de cada muestra es mayor que en pathtracing y se requiere el tiempo de combinación de los caminos.

Igual que en pathtracing, la imagen tiene ruido a menos que se utilice un alto número de muestras.

Transporte de Luz de Metrópolis

Propuesto en 1997 por Veach y Guibas [VEA97], utiliza la información sobre el espacio en la escena de una forma más eficiente. Es un método diferente a Pathtracing, ya que no realiza muestras aleatorias para resolver el valor de la integral, sino que genera una distribución de muestras que es proporcional a la función que queremos resolver (esto es, la densidad de rayos es proporcional a la radiancia, concentrando rayos en las regiones más brillantes de la imagen).

Este método comienza con una distribución de muestras aleatorias (usando pathtracing bidireccional) de todas las fuentes de luz de la escena. Estos caminos son clonados aleatoriamente y mutados. Si el path mutado es inválido (por ejemplo, atraviesa un muro), entonces se descarta, y el camino original se utilizará de nuevo para obtener una nueva mutación. Si el camino es válido, entonces será aceptado con una determinada probabilidad.

Ventajas

Este método funciona bien para simular configuraciones de luz que son difíciles para otros métodos (por ejemplo, un agujero abierto en una pared donde existe una pequeña luz). El método de Metrópolis encuentra el camino a través de mutaciones del camino, siendo mucho más eficiente que Pathtracing o Pathtracing bidireccional.



Figura 5. *Escena renderizada con diferentes métodos de render.*

Desventajas

En escenas *"normales"*, no existe ganancia frente a otras técnicas como Pathtracing o Pathtracing Bidireccional.

Comparación de tiempos en una escena

En la siguiente tabla se muestran los tiempos de render de la escena de la figura 5. La resolución de todos los ejemplos es de 640x480 píxeles y oversampling de 8 muestras por píxel. La escena está formada por 51016 polígonos. El modelo del dragón tiene propiedad de reflexión (2 niveles de recursión), y las copas de reflexión (3 niveles) y refracción (4 niveles).

La imagen ha sido renderizada en un Centrino 2GHz, con 1 GB de RAM bajo Debian GNU/Linux.

Método	Parámetros	Tiempo
ScanLine	**Nota:** *En la imagen de la figura no se ha variado la propiedad del material para que se simule la transparencia con Ztransp, pero se podría realizar con un tiempo de render similar.*	5 Seg.
Raytracing	**Nota:** *Todos los ejemplos se han renderizado con Oversampling de nivel 8. La variación del nivel de oversampling en una misma configuración de resolución del Octree (256) puede suponer las siguientes diferencias de tiempo:* **AA (0):** 10" **AA (5):** 46" **AA (8):** 74" **AA (11):** 102" **AA (16):** 146"	Resol. Octree (64): 1:34. Resol. Octree (128): 1:16. Resol. Octree (256): 1:14. Resol. Octree (512): 1:23.
Raytracing + Ambient Occlusion	**10 Muestras** · **Distancia** para aplicar AO: 10 **Energía global AO:** 2.0 **Nota:** *La variación en el número de muestras es determinante en el tiempo de render. Con una resolución del Octree de 128, los tiempos de render son:* **2 M:** 129" **5 M:** 218" **8 M:** 426" **10 M:** 615 "	Resol. Octree (64): 14:02. Resol. Octree (128): 10:15. Resol. Octree (256): 11:22. Resol. Octree (512): 16:17.
Pathtracing (Skydome)	Profundidad rayo máxima desde la cámara: 9 Emit Power: 1.5 AutoAA **Nota:** *La diferencia se establece en el número de samples de la Hemilight.*	Low (16 Samples): 9:53. Medium (24 Samples): 10:23. High (36 Samples): 16:28. Higher (64 Samples): 27:37. Best (128 Samples: 39:05.
Pathtracing (Full) + Irradiance Cache	Sin refinamiento de sombras. Sin mapeado de fotones. Calidad de Sombras: 0.9 Emit Power: 1.5 GI Power: 0.5 **Nota:** *La diferencia se establece en el número de samples de la Pathlight.*	Low (128 Samples): 29:41. Medium (256 Samp.): 40:30. High (512 Samples): 53:12. Higher (1024 Samples): Best (2048 Samples):

Bibliografía

[BOU70] Bouknight, W. J. , *"A procedure for generation of three-dimensional half-tone computer graphics"*, Communications of the ACM (1970)

[COH85] Cohen, M. F., Greenberg D. P. , *"The Hemi-Cube: A radiosity solution for complex environments"*, Siggraph 85 (Proceedings) (1985)

[GOR84] Goral, C. Torrance, K.E., Greenberg, D.P. Battaile, B. , *"Modelling the interaction of light between diffuse surfaces"*, Proceedings of SIGGRAPH (1984)

[JEN01] Jensen H.W. , *"Realistic Image Synthesis Using Photon Mapping"*, , A.K.Peters, (2001)

[JEN96] Jensen, H. W. , *"Global Illumination using photon maps"*, Eurographics Rendering Workshop (1996)

[KAJ86] Kajiya, J.T. , *"The rendering equation"*, Computer Graphics (1986)

[COO86] Robert L. Cook , *"Stochastic sampling in computer graphics"*, ACM Transactions on Graphics (1986)

[VEA97] Veach, E., Guibas, L.J. , *"Metropolis Light Transport"*, Proceedings of SIGGRAPH '97 (1997)

[WAR92] Ward, G. J., Heckbert, P. , *"Irradiance Gradients"*, Third Eurographics Workshop on Rendering (1992)

[WHI80] Whitted, T. , *"An improved illumination model for shaded display"*, Communications of the ACM (1980)

[ZHU98] Zhukov, S., Iones, A., Kronin, G. , *"An ambient light illumination model"*, Proc. of Eurographics Rendering Workshop '98 (1998)

[LAF93] Lafortune, E.P., Willems, Y.D. *"Bidirectional Path tracing"*, Compugraphics (1993)

[COO84] Cook, R.L, Porter, T., Carpenter, L., *"Distributed ray tracing"*, Computer Graphics (1984)



Sección V

Desarrollo

Fotograma de la Película de Animación *"Elephants Dream"* realizada con Blender

```
def trazar (r, prof):
   c = color()

   for obj in lObjetos:    # Probam   on todos los obj
      obj.intersecta(r)

   if (r.objInter != None):
      matIndex = r.objInter.m
      vInterseccio    r.orige    (r.direccion    .disInter)
      vIncidente         ecc on - r origen
      vVueltaOrigen = r        on
      vVueltaOrigen   rmal
      vNormal = r obj   er.g        Interseccion)
      for luz in l  ces:
         if (luz.tipo   'amb
            c += luz.color
         elif (luz.tipo == 'puntual')
            dirLuz = luz.posicion - vInt        ion
            dirLuz.normalizar()
            rayoLuz = rayo(vInterseccion,  irLuz)
            sombra = calculaSombra (rayoL  , r.objInt  )
            dirLuz = luz.posicion - vInte  eccion # Si  normalizar
            dirLuz /= dirLuz.modulo()  #  plicamos el   ll-off
            NL = vNormal.pEscalar(dirLuz
            if (NL > 0):
               if (lMateriales[matIndex].        >      # ------- Difuso
                  colorDifuso = luz.color * lMateriales[matIndex].cDifuso * NL
                  colorDifuso.r *= lMateriales[matIndex].color.r * sombra
                  colorDifuso.g *= lMateriales[matIndex].color.g * sombra
                  colorDifuso.b *= lMateriales[matIndex].color.b * sombra
                  c += colorDifuso
               if (lMateriales[matIndex].cEspecular > 0): # ----- Especular
                  R = (vNormal * 2 * NL) - dirLuz
                  espec = vVueltaOrigen.pEscalar(R)
                  if (espec > 0):
                        espec = lMateriales[matIndex].cEspecular * \
                                pow(espec, lMateriales[matIndex].dEspecular)
                        colorEspecular = luz.color * espec * sombra
                        c += colorEspecular
      if (prof < profTrazado):
         if (lMateriales[matIndex].cReflexion > 0):   # -------- Reflexion
            T = vVueltaOrigen.pEscalar(vNormal)
            if (T>0):
               vDirRef = (vNormal * 2 * T) - vVueltaOrigen
               vOffsetInter = vInterseccion + vDirRef * PEQUENO
               rayoRef = rayo(vOffsetInter, vDirRef)
               c += trazar (rayoRef, prof+1) * lMateriales[matIndex].cReflexion
         if (lMateriales[matIndex].cTransmitividad > 0):  # ---- Refraccion
            RN = vNormal.pEscalar(vIncidente * -1)
            vIncidente.normalizar()
            if (vNormal.pEscalar(vIncidente) > 0):
               vNormal = vNormal * -1
               RN = -RN
               n1 = lMateriales[matIndex].iRefraccion
               n2 = 1
            else:
               n2 = lMateriales[matIndex].iRefraccion
               n1 = 1
            if (n1!=0 and n2!=0):
               par_sqrt = sqrt (1 - (n1*n1/n2*n2)*(1-RN*RN))
               vDirRefrac = vIncidente + (vNormal * RN) * (n1/n2) - (vNormal * par_sqrt)
               vOffsetInter = vInterseccion + vDirRefrac * PEQUENO
               rayoRefrac = rayo(vOffsetInter, vDirRefrac)
               c += trazar (rayoRefrac, prof+1) * lMateriales[matIndex].cTransmitividad
   return c
```

The soul of the monkey

YafRayNet: hardware a medida para renderizar con YafRay

Roberto Domínguez *(Hardware y Software)*
Miguel Ángel Guillén (Hardware y Software)
Guillermo Vayá *(Software)*



Este proyecto está basado en Yafray (www.yafray.org), uno de los motores de render integrados en Blender. La calidad de YafRay alcanza niveles profesionales, y por tanto requiere muchos recursos cuando se intenta afrontar un proyecto de gran envergadura.

Si planificamos un proyecto profesional, tenemos por un lado el diseño artístico, modelado, texturizado, iluminación y animación, y luego además el tiempo de render, que puede ser tanto o más que todo lo anterior junto. Así que nos pusimos a pensar cómo evitar tener nuestras máquinas ocupadas durante semanas o incluso meses haciendo un render.

La solución en la que estamos trabajando consta de tres partes:

1. Un hardware específico para renderizar con YafRay. Funcionalmente una caja negra a la que envías la escena en xml y la renderiza.
2. Un sistema escalable desde el punto de vista del usuario, de manera que pueda usar tantas tarjetas a la vez como quiera. Para conseguir esto, hemos hecho las tarjetas externas al PC y conectadas por red. Se puede tener una granja de render montando una red de área local donde el servidor es el PC y los clientes las tarjetas YafRayNet.
3. Un sistema escalable entre usuarios. El software servidor que está corriendo en tu PC puede compartir los recursos de cálculo de tus tarjetas con otros usuarios en una red P2P, tipo Edonkey o Emule. Así también puedes usar las tarjetas de otros además de la tuya.

El proyecto tiene la primera parte, lo que es el diseño del hardware y todo el software de control de la tarjeta, prácticamente terminado y en fase de pruebas.

TARJETA YAFRANET

La tarjeta se compone principalmente de un ColdFire de Motorola y 4 micros de procesamiento de señal digital de Texas. El funcionamiento es el siguiente:

1. La tarjeta se enciende. Hemos desarrollado un cargador de arranque que hemos instalado en una pequeña flash conectada al ColdFire. El cargador arranca, inicializa el hardware, comprueba que todo funciona perfectamente, activa el Phyter, que es el micro que controla la Ethernet, la FPGA que controla el canal DMA y la comunicación con los dos DSPs, y atiende al WatchDog de temperatura. Queda finalmente corriendo la pila TCP y se queda a la espera de una respuesta por parte de la aplicación servidora en el PC.
2. El servidor le envía a la tarjeta por TFTP un archivo comprimido. En este archivo hay varias cosas:
   - Un kernel de linux, basado en uCLinux y al que le hemos desarrollado los drivers de todo el hardware que lleva la placa. La parte más difícil ha sido los drivers de DMA para conectar los DSPs al ColdFire, ya que éste sólo tiene un canal

DMA, y aún tenemos algunos problemas de equalización de pistas que nos hace perder un poco de rendimiento.

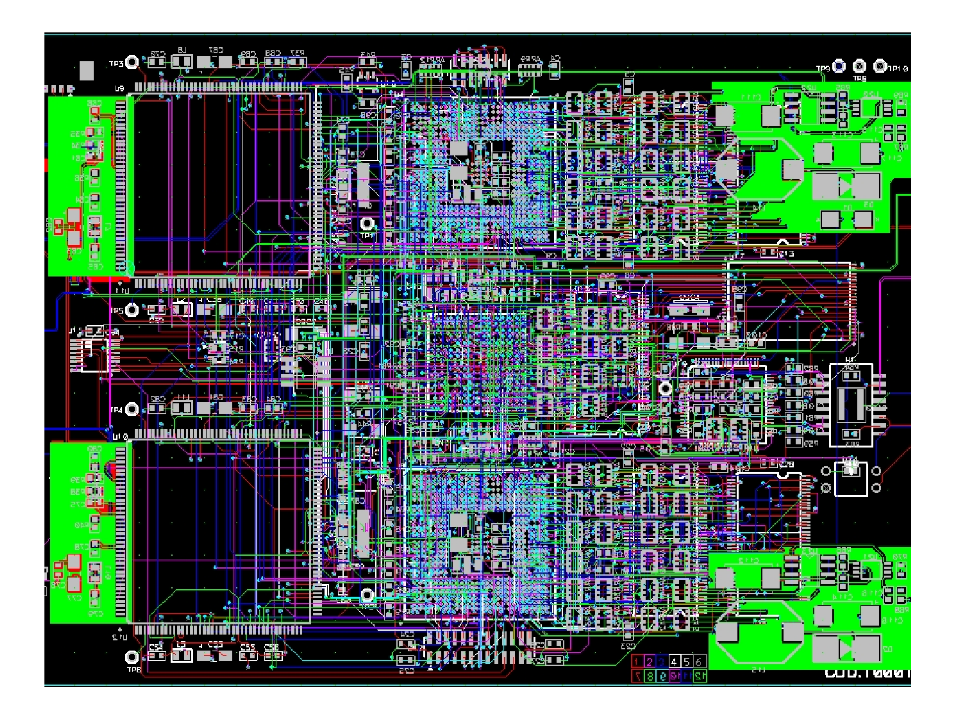
- Un sistema de archivo para crear un disco duro virtual en memoria SDRAM, que lleva una estructura típica de Linux.
- La aplicación cliente de renderizado.
- Y el motor de render que tendrá que cargarse en cada uno de los DSPs.

Todo esto, sistema operativo y aplicaciones nos ocupa 2MB.

3. El ColdFire recibe el archivo, lo descomprime en SDRAM, ejecuta el kernel de linux, monta el disco virtual, y ejecuta el software cliente. El tiempo total de arranque es de unos 3 segundos. La ventaja de hacerlo así, en vez de tenerlo todo grabado en la memoria Flash, es que en cualquier momento se puede actualizar el software del PC, y con ello se actualiza tanto el sistema operativo que corre en el ColdFire como el programa que corre en los DSPs. De hecho el programa de los DSPs puede ser cualquier aplicación que requiera una gran potencia de cálculo en coma fija. Las primeras pruebas que hemos hecho para hacer los drivers y medir el rendimiento ha sido con un compresor en tiempo real de MPEG4.
4. Cuando el servidor del PC envía un trabajo para ser renderizado, el ColdFire carga YafRay en la memoria interna de los DSPs, carga la escena que tenemos que renderizar en la SDRAM externa de cada DSP, y arranca los micros. Para ello cada DSP se conecta al ColdFire por DMA sobre un bus HPI, un bus muy similar al PCI de los PCs, a 66Mhz.
5. Los DSPs renderizan su parte de la escena y cuando terminan devuelven el render al ColdFire y éste al servidor del PC.

Los DSPs que estamos montando son micros de Texas de la serie 6000, TMS320DM64x, a 1 Ghz. Cada micro tiene 8 unidades de cálculo de coma fija, con lo que tenemos una potencia de 8000 MIPS de 32 bits por micro. Un Pentium IV a 1 Ghz con MMX o SSE hace 4 mil millones de instrucciones por segundo, por lo que la tarjeta viene a ser en coma fija el equivalente a un Pentium IV a 8 Ghz.

El proceso lo va a hacer bastante más rápido de lo que lo haría un PC de gama alta, y además deja el PC libre para seguir trabajando con Blender o lo que se quiera. Como las tarjetas se conectan por ethernet se pueden poner hasta 255 conectadas con Hubs. Lo que tenemos es una red local de PCs de gama alta que cuestan unos 200 euros cada uno.



Diseño hardware

La placa es bastante compleja, de 12 capas, con microvías y memorias BGA, montada a mano y rutada con Espectra y Picad. En general hemos utilizado componentes de bajo consumo para evitar que se nos dispare el tamaño de la tarjeta. Por ahora en pruebas consume a pleno rendimiento unos 0.6A a 12V. Para que no se queme le hemos añadido un sensor de temperatura y un Watchdog. La memoria es de un solo puerto por razón de coste, lo que nos ha obligado a gestionar bloqueos entre los DSPs y el ColdFire para evitar que accedan a la memoria simultáneamente. También hemos añadido un estabilizador de tensión que hace que los micros no arranquen hasta que la tensión es estable, y como es controlable desde el ColdFire nos permite también tener reseteados los DSPs hasta que el ColdFire haya configurado el bus HPI.

Una de las etapas más complicadas es portar YafRay a la arquitectura del DSP.

Porting de YafRayNet a coma fija

La tarjeta de YafRayNet, utiliza un procesador coldfire para realizar los cálculos. El problema que genera el uso de este procesador es el hecho de que utiliza matemáticas de coma fija, al contrario que la mayoría de los ordenadores de uso común estos días que usan matemáticas de coma flotante.

Un valor en coma fija, es aquel que tiene un número fijo de dígitos decimales, es decir, valores cuya precisión es constante. Si un tipo de datos tiene una precisión de 0,001 (3 dígitos decimales) un valor como 0,0002 sería considerado como 0,000; ademas si el numero es muy grande, la coma fija tampoco da soporte al valor, produciéndose un desbordamiento y dando un valor que no tiene nada que ver con el que se quiere conseguir. Como se puede apreciar, esto puede dar problemas a la hora de generar números muy pequeños o muy grandes, ya que no puede alcanzarlos. Por el contrario, la coma flotante, no tiene una precisión fija, ya que consta de dos partes una es el numero (llamado mantisa) y el otro es una potencia (llamandose radix a la base). De esta forma, podemos generar numeros como $1 \cdot 10^{-3}$, lo que nos da el valor de 0,001. De igual manera se pueden representar numeros grandes si usamos un exponente positivo. He de hacer notar, que los computadores no usan un radix de 10, sino generalmente una potencia de 2.

Esta diferencia en la arquitectura de los procesadores plantea la necesidad de un *port* del proyecto yafray para ser usado en la placa YafRayNet. Ya que es muy dificil conocer de antemano los valores que va a manejar. Una escena muy grande o una muy pequeña podrian causar errores de precisión como los vistos anteriormente.

Por todo lo anterior se ha decidido seguir los siguientes pasos a la hora de portar el codigo:

1. Realizar una version modificada de yafray para ordenadores personales, cambiando el tipo de datos de float (el nombre que se usa para indicar coma flotante) por un tipo de datos que emule el comportamiento de la coma fija.
2. Ofrecer esta versión a la comunidad para que la prueben y encontrar posibles errores tanto de precisión como otros que no se hayan considerado. Esta versión sera claramente mas lenta, ya que no es un tipo de datos nativo, pero dado que es una version para realizar busqueda de problemas potenciales no se ha considerado la velocidad como un factor importante.

3. Una vez visto que no hay problemas o que los problemas existentes son facilmente resolubles, se procederá a portar el codigo al ensamblador propio del coldfire.
4. Fase de depuración y optimización del codigo.

He de recalcar, que la fase 2 es crítica, ya que es aqui donde se ha de decidir como se va a realizar el *"porting"*. Es importante que los "beta-testers" provean de toda la información posible, a fin de sacar todos los problemas, por pequeños que estos sean. Ademas, el resultado de esta fase, mostrará si la conversión del codigo sera una tarea facil o si habra que recrear una coma flotante ficticia para que todo funcione correctamente. Espero que esta tarea no sea necesaria, no solo por el tabajo que ello implicaria, sino tambien porque aumentaria el tiempo de proceso de forma importante.

Exportación desde Blender

Fernando Arroba Rubio
gnotxor@gmail.com ::



Este trabajo pretende exponer de una manera sencilla y amena los métodos de exportación de objetos desde Blender a cualquier formato externo. Como ejemplo ilustrativo se ha seleccionado el lenguaje VRML v2.0 por dos motivos fundamentales: Es un lenguaje muy extendido e independiente del sistema operativo de la máquina, pero principalmente es un lenguaje de descripción de espacios virtuales comprensible para los humanos. Los ficheros se guardan en formato de texto y se pueden modificar y crear con cualquier editor de texto.

El trabajo hace hincapié en la estructura que debe tener un script exportación desde Blender, aunque algunas de las apreciaciones son subjetivas. Para no desviar la atención sobre el eje central del mismo no se tratan temas como la "herencia" cuando habla de las clases en la exportación. De la misma manera tampoco se hacen referencias exhaustivas al lenguaje VRML para el cual es recomendable consultar algún tipo de documentación propia. Tampoco se ha pretendido hacer un repaso exhaustivo de las matemáticas que envuelven al 3D y en lo posible se obviarán de la misma manera que se ha intentado evitar complejidades de este en el script.

Entrando en el menú sin hacer mucho ruido

El primer punto importante para que Blender reconozca un script es la primera línea del código. Paralelamente a lo que hacen los script de python normales, que lanzan el intérprete, Blender lanza el suyo propio:

```
1    #!BPY
```

A continuación de esta línea podemos añadir un comentario del script que añada funcionalidad a dicha identificación. Es decir, podemos hacer un registro del mismo en los menús de Blender.

```
2   """Registramos el script en los menús de Blender
3   Name: 'Ejemplo de la conferencia...'
4   Blender: 234
5   Group: 'Export'
6   Tooltip: 'Este es el ejemplo para la conferencia'
7   """
```

En este comentario especificamos el texto que aparecerá en el menú (3), la versión para la que está escrito (4), el grupo al que pertenece el script (5) y la etiqueta de ayuda que aparece cuando situamos el cursor sobre la entrada del menú (6).

Si la versión de Blender sobre la que corremos el script es anterior a la que marca la cabecera nos mostrará un aviso de que podría no funcionar correctamente:

```
Version mismatch: script was written for Blender 234.
It may fail with yours: 232Version mismatch: script
was written for Blender 234. It may fail with yours:
232
```

Es posible añadir más información a dicha cabecera, como el poder dar opciones al usuario a la hora de exportar. Por ejemplo, si queremos que el usuario pueda seleccionar si quiere exportar toda la escena o sólo aquellos objetos que se encuentren seleccionados, podríamos añadir submenús de la siguiente manera:

```
8   """Registramos el script en los menús de Blender
9   Name: 'Ejemplo de la conferencia...'
10   Blender: 234
11   Group: 'Export'
12   Submenu: 'Todos los objetos...' todos
13   Submenu: 'Objetos seleccionados...' selec
14   Tooltip: 'Este es el ejemplo para la conferencia'
15   """
```

Si utilizamos submenús (12-13) debemos capturar el valor de los mismos (todos o selec). Como no todas las versiones de Blender pueden capturar ese argumento es mejor tener algún tipo de código que controle las excepciones que puede provocar. Vamos a ver el ejemplo, esta vez con el código completo:

```
16   """Registramos el script en los menús de Blender
17   Name: 'Ejemplo de la conferencia...'
18   Blender: 234
19   Group: 'Export'
20   Submenu: 'Todos los objetos...' todos
21   Submenu: 'Objetos seleccionados...' selec
```

```
22  Tooltip: 'Este es el ejemplo para la conferencia'
23  """
24  # ---------
25  # MAIN
26  # ---------
27  try:
28    ARG = __script__['arg'] # Capturo el argumento del
                                submenú
29  except:
30    print "Versión antigua"
31    ARG = 'selec'
32
33  print "Se ha seleccionado la opción: %s" % ARG}
```

Lo más interesante a destacar en el código es el manejo de excepciones (27-31) que se hace para cargar en una variable global ARG (28) la selección del usuario. Al manejar la excepción podemos mantener en funcionamiento el script y así poder asignarle un valor por defecto a la variable (31). En la linea 33 podemos observar que mostramos a través de la consola ese argumento como una cadena de caracteres más. Es mucho más interesante, por supuesto, si utilizamos dicho valor para hacer cosas distintas, por ejemplo:

```
34  if ARG == 'todos':
35     losObjetos = Scene.getCurrent().getChildren()
36  else:
37     losObjetos = Blender.Object.GetSelected()
```

Es decir, si el argumento pasado es “todos” se cargan todos los objetos de la escena actual en una lista llamada “losObjetos”, mientras que si el argumento es otro, se cargarán solo los objetos que estén seleccionados. Recordamos que si se produce la excepción que no permite cargar el argumento desde el submenú, el script cargará por defecto sólo los objetos seleccionados.

Por otro lado, es muy recomendable que tengamos algún código que controle la versión de Blender que se está ejecutando. Aunque ya hemos comentado que podemos (debemos) establecer la versión de Blender para la que se ha hecho el script y que el propio programa tiene un cierto control sobre ello, no está de más que añadamos más seguridad a nuestro script de forma manual. Esto evitará problemas y sinsabores al usuario de una forma sencilla y elegante. Por ejemplo, si sabemos que nuestro script no funciona en alguna versión determinada podríamos avisar al usuario para evitar que pruebe una y otra vez sin conseguir nada. Es frustrante cuando lanzas un script y no hace lo que esperas y además tampoco sabes el porqué. Por ejemplo, si sabemos que estamos utilizando características que

aparecieron en el API a partir de la versión 2.30 podríamos escribir un poco de código de la siguiente manera, y comenzar realmente el procesamiento del fichero a partir de la línea 42:

```
38  if Blender.Get('version') < 230:
39     print "Aviso: ¡Versión incorrecta de blender!"
40     print "No está utilizando Blender 2.30 o superior,
              descargue"
41     print "una versión más actualizada desde
              http://blender.org/"
42  else:
43      .....
```

Un vistazo a lo que necesitamos

Antes de comenzar a detallar algunos módulos que debemos emplear de forma casi sistemática, necesitamos comprender un poco cómo está organizada la información que queremos exportar. También debemos tener claro el formato en el que debe ser escrita. Esto en algunas ocasiones (por no decir la mayoría) no coincide. Para hacer una exportación limpia muchas veces deberemos tener en cuenta esas peculiaridades y adaptarlas lo mejor posible.

Un poco de matemáticas

Un primer foco de problemas puede venir de la geometría de los objetos. En algunas ocasiones lo podremos reducir a una cuestión de equivalencias de escala, en otras necesitaremos un conocimiento un poco más preciso de las matemáticas que envuelven el 3D.

Para empezar vamos a echar un vistazo a la matriz de los objetos. Comenzamos para ello con la representación de un punto. Un punto no tiene dimensión, simplemente es una posición (coordenadas) en un espacio 3D. Sin embargo, la misma posición puede estar representada por valores distintos dependiendo del origen de coordenadas que utilicemos. Muchos sistemas y programas que representan figuras en un espacio 3D siguen pautas parecidas: almacenan la descripción de una malla con respecto a un punto (que llamaremos centro) y añaden una matriz de modificación. En esa matriz se guardan informaciones tan valiosas como la orientación, posición y escalado de la malla. Por último, para conocer la posición real de un punto de esa malla sólo hay que aplicar dicha matriz a las coordenadas relativas de dicho punto dentro de la malla.

Puede parecer algo alejado del usuario normal, pero no lo es, basta pulsar "n" en modo objeto para que blender nos muestre los valores de la matriz del objeto seleccionado (aunque no la matriz misma).

Otro tema es la orientación de los ejes de coordenadas. Concretamente si son ejes de coordenadas "*a derechas*" o "*a izquierdas*". Blender utiliza un sistema orientado a derechas, es decir, "x" crece hacia la derecha, "y" crece hacia arriba y "z" crece hacia afuera de la pantalla. Otros programas, como POV-Ray utilizan un sistema orientado a izquierdas, es decir, "x" e "y" están orientados a derecha y arriba, respectivamente y "z" crece hacia adentro de la pantalla.

Hay que tener en cuenta todos estos detalles a la hora de exportar a otros sistemas, pues sus valores pueden representarse de otra manera. Antes debíamos realizar todos los cálculos necesarios por nosotros mismos (producto escalar, producto de vectores, cálculo del módulo de un vector, etc). En la actualidad contamos con un módulo de utilidades matemáticas dentro de blender: Mathutils.

Blender (la madre del cordero)

Este módulo es el base de todos los demás. Por otro lado, también proporciona información útil a la hora de la exportación. Por ejemplo, si queremos proporcionar al usuario un nombre de fichero donde guardar la exportación, podemos solicitar a Blender un nombre de fichero.

```
44   nombreFichero = Blender.Get('filename')
```

Guarda en la variable "nombreFichero" el nombre del último archivo que se ha abierto o guardado desde blender. Otra de las informaciones útiles que proporciona este módulo es la versión de blender que está en funcionamiento.

```
45   blenderVer = Blender.Get('version')
```

A partir de este módulo se harán las importaciones.

```
46   import Blender
47   from Blender import NMesh, Window
48   from Blender import Scene, Mathutils
49    ....
```

Es aconsejable utilizar importaciones desde el módulo principal y no hacer importaciones globales con el comodín “*” para no llenar el diccionario del espacio de nombres de python con demasiadas entradas, muchas de ellas inútiles.

Por otro lado, el obligarnos a utilizar el nombre del módulo al que pertenece alguna función, o clase, hará más legible el código que escribamos, sobre todo a la hora de revisiones posteriores.

Facilitando la vida al usuario

Hay ocasiones en que el programador hace el script para uso propio. Esto conlleva la tendencia a interactuar con él desde el propio código, sin embargo, hace prácticamente imposible que un tercero utilice dicho script.

En otras ocasiones estamos obligados a pensar en cómo otra persona, que no tienen porqué saber nada sobre programación, conseguirá aprovechar el trabajo que realiza el script. Para facilitarnos el trabajo, Blender provee de dos módulos interesantes: *Sys* y *Window*. En nuestro ejemplo utilizamos tan sólo uno de ellos para preguntar el nombre del fichero donde guardará la salida y comenzará el proceso al pulsar el botón correspondiente.

```
50  Window.FileSelector(fichero, "Exportar")
```

Esta línea de código muestra en la ventana de Blender un diálogo de fichero. El primer argumento de *FileSelector* es la función que se ejecutará si se pulsa el botón del diálogo; a dicha función le pasará el nombre del fichero seleccionado. El segundo argumento es el texto que aparecerá en dicho botón. Lo que nos interesará, por supuesto, es la estructura de la función que realiza el trabajo.

```
51  def fichero(nomFichero):
52     if nomFichero.find('.wrl', -4) < 0:
53       nomFichero += '.wrl'
54     # Abro el fichero y lo conecto con std
55     f = open(nomFichero, 'w')
56     std = sys.stdout
57     sys.stdout = f
58     # Proceso los datos
59     sce = ExportadorEscena(nomFichero)
60     sce.proceso()
61     # Devuelvo el control de stdout y cierro el fichero
62     sys.stdout = std
63     f.close()
```

Lo primero que hace dicha función es comprobar el nombre del fichero que nos proporciona el selector de ficheros de Blender. Principalmente la extensión, hacemos esto para evitar pérdidas de datos producidas por una selección errónea por parte del usuario. Es posible que al seleccionar el nombre de un fichero éste marque equivocadamente el fichero .blend de origen, o cualquier otro. De esta manera, si el nombre del fichero no concluye con la extensión deseada, esta será añadida salvaguardando de la sobrescritura datos ajenos al script y el proceso que representa.

En las siguientes líneas (54-63) se realiza todo el proceso. Básicamente en tres pasos:

1. Se abre un fichero y se redirecciona la salida estándar a él (55-57).
2. Se realiza el proceso de exportación (59, 60).
3. Se devuelve la salida estándar a su estado original (62,63).

Esto facilita en gran medida la legibilidad del código pues simplemente tendremos que utilizar instrucciones *print* para realizar la exportación. En contrapartida, hace necesaria la instalación de Python en el sistema pues utiliza su módulo *sys*.

Otra dificultad añadida aparece si el formato al que queremos exportar el fichero es binario. En este caso no podríamos utilizar esta forma de simplificación necesitando además el módulo *struct* que Python proporciona para esos menesteres.

Los objetos vistos desde Blender

No es objeto del presente trabajo hacer una pormenorizada explicación de los distintos módulos que provee Blender para la automatización de tareas. Para eso remitimos al lector a la documentación correspondiente. Sin embargo, no podremos avanzar mucho sin unas pocas palabras sobre el trabajo fundamental con algunos de estos módulos.

El objetivo del script de ejemplo es la “pedagogía” más que la eficiencia o la exactitud, por lo que se ha intentado mantener en un mínimo razonable todo el aparato de importaciones y código extraño. Incluso el autor se ha reprimido del uso de la herencia a la hora de plantearlo. Por ello, sólo haremos hincapié en los puntos básicos. Uno de los cuales es el bucle de exportación, que en nuestro caso se encuentra dentro de la clase *ExportadorEscena*, concretamente *ExportadorEscena.proceso()*.

```
64  class ExportadorEscena:
65   def __init__(self, nomFichero):
66     self.fichero = nomFichero
67
68   def proceso(self):
69     global losObjetos    # Utilizo los objetos cargados
                              en la lista global
70     print "#VRML V2.0 utf8\n"   # cabecera del fichero
71     print "#%s" % self.fichero  # El nombre del fichero
72     for obj in losObjetos:      # Proceso la lista de
                                     objetos
73       tipo = obj.getType()   # Obtener el tipo de
                                  objeto que vamos a tratar
74       if tipo == 'Mesh':
75          o = Objeto(obj)
76          o.proceso()
77       elif tipo == 'Camera':
78          c = Camara(obj)
79          c.proceso()
80       elif tipo == 'Lamp':
81          luz = Luz(obj)
82          luz.proceso()
```

Lo primero que haremos en el proceso será escribir la cabecera del fichero (70) y añadimos un comentario al mismo con el nombre del fichero (71). Obsérvese que basta con escribir directamente en la salida estándar, puesto que la habíamos redireccionado a nuestro fichero. Después de esta cabecera comienza un bucle *for* que constituye el centro de todo el script.

Dicho bloque (72-82) se encarga de procesar los objetos que se hayan incluido en la lista global *losObjetos* que vimos al principio. Puesto que en el fichero .blend que nos servirá de ejemplo sólo hay objetos de tipo *Mesh*, *Camera* y *Lamp*, el script sólo tiene en cuenta estos tipos. Se puede observar la similitud en la estructura de dicho proceso: primero se crea un objeto del tipo conveniente y a continuación se llama a su función miembro *proceso()*. Esta coincidencia está llamada a ser tratada mediante herencia, sin embargo, se ha evitado llevarla a cabo en este caso para no confundir a aquellos que no estén acostumbrados a la programación orientada a objetos. La ventaja de dicha programación nos permitiría, por ejemplo, escribir sólo una vez el código que realizara el cambio de sistema de coordenadas en la clase base y que los demás objetos hijos lo heredaran.

Los objetos desde VRML

La estructura que siguen las clases dentro del script están acomodadas a la estructura que debemos generar para VRML. Es decir, los objetos tienen sus mallas definidas como listas de vértices que luego son referenciados en listas de caras. Dentro de esa estructura de sólido llevan también incluida una estructura de "apariencia".

En el caso de la apariencia, y con el objeto de abreviar un poco, sólo se han tenido en cuenta los parámetros más importantes del material y se han obviado otros como las texturas.

El problema fundamental desde VRML era la representación de objetos y el cambio de origen para los puntos que definen la malla. Como sabemos, Blender almacena la información de las mallas en coordenadas locales. Es decir, las coordenadas de un punto están dadas teniendo como origen el "centro" del objeto al que pertenece. Para saber la posición real de un punto (desde el 0 absoluto) debemos tener en cuenta las coordenadas del punto respecto al centro del objeto y la posición, rotación y tamaño del objeto. Al pasar esta información a VRML nos damos cuenta que un sistema no se corresponde con el otro. Las transformaciones en VRML se almacenan de forma distinta y en este sistema no es lo mismo rotar una figura y luego trasladarla, que trasladar una figura y luego rotarla, pues las transformaciones se realizan todas desde el centro absoluto.

Ante estas consideraciones, la salida más sencilla era exportar la malla de forma absoluta haciendo la conversión de "centros".

```
83  class Malla:
84   ....
85   def absoluto(self, v):
86     vr = Mathutils.Vector([v[0], v[1], v[2]])
87     vr.resize4D()
88     return Mathutils.VecMultMat(vr, self.matriz)
```

El método *absoluto()* se encarga de convertir una lista de tres valores en un objeto *vector* del módulo *Mathutils* para luego redimensionarlo y multiplicarlo por la matriz propia del objeto. Devuelve un vector con la trasformación realizada.

Reutilizando cosas

Otro de los aspectos a tener en cuenta cuando se realice una exportación es la posible reutilización de objetos, texturas, materiales... que pueda realizar el sistema al que estamos exportando. No sólo eso, sino también en qué manera lo vamos a hacer.

En nuestro caso hemos decidido reutilizar los materiales. Para ello hemos habilitado un diccionario python donde almacenar los que vamos encontrando. Eso nos permite luego decidir si tenemos que hacer la definición completa o sólo una inclusión de la misma.

```
89   materiales = {}
90   class Material:
91    def __init__(self, mat):
92      global materiales
93      self.nombre = mat.getName()
94      if materiales.has_key(self.nombre):
95        self.definido = 1
96      else:
97        self.definido = 0
98        materiales[self.nombre] = 1
99        self.color = mat.getRGBCol()
100         self.ambiente = mat.getAmb()
101         self.spec = mat.getSpecCol()
102         self.emit = mat.getEmit()
103         self.brillo = mat.hard/255.0
104         self.transparencia = 1 - mat.alpha
```

En el fragmento de script se puede destacar la definición de un diccionario python vacío llamado *materiales* (89). A continuación, dentro de la clase que va a procesar este tipo de objetos activamos *definido* como flag que nos dirá si hay que hacer el proceso completo o no (94-97). Si no está definido con anterioridad el constructor de la clase añade ese material al diccionario (98) y carga los valores necesarios para procesar el material (99-104).

El script completo

```
105  #!BPY
106  """ Registramos el script en los menús de Blender
107  Name: 'Ejemplo de la conferencia...'
108  Blender: 23
109  Group: 'Export'
110  Submenu: 'Todos los objetos...' todos
111  Submenu: 'Objetos seleccionados...' selec
112  Tooltip: 'Esto es un ejemplo para la conferencia'
113  """
114
115  import Blender
116  import sys
117  from Blender import NMesh, Lamp, Window
118  from Blender import Scene, Mathutils
119
```

```
120  losObjetos = []
121  materiales = {}
122
123  class Material:
124   def __init__(self, mat):
125     global materiales
126     self.nombre = mat.getName()
127     if materiales.has_key(self.nombre):
128       self.definido = 1
129     else:
130       self.definido = 0
131       materiales[self.nombre] = 1
132       self.color = mat.getRGBCol()
133       self.ambiente = mat.getAmb()
134       self.spec = mat.getSpecCol()
135       self.emit = mat.getEmit()
136       self.brillo = mat.hard/255.0
137       self.transparencia = 1 - mat.alpha
138
139   def proceso(self):
140     if self.definido == 1:
141       print "\t\t# Material:", self.nombre, "ya definido."
142       self.escribeMaterialRepetido()
143     else:
144       print "\t\t# Definición del material:", self.nombre
145       self.escribeMaterial()
146
147   def escribeMaterial(self):
148     # Inicio la definición del material
149     print "\t\tmaterial DEF %s Material {" % self.nombre
150     print "\t\t\tdiffuseColor", self.color[0],
             self.color[1], self.color[2]
151     print "\t\t\tambientIntensity", self.ambiente
152     print "\t\t\tspecularColor", self.spec[0], self.spec[1],
             self.spec[2]
153     print "\t\t\temissiveColor", self.emit, self.emit,
             self.emit
154     print "\t\t\tshininess", self.brillo
155     print "\t\t\ttransparency", self.transparencia
156     # Cierro la definición del material
157     print "\t\t}", "# Final del material", self.nombre
158
159   def escribeMaterialRepetido(self):
160     print "\t\tmaterial USE", self.nombre
161
162  class Apariencia:
163   def __init__(self, m):
164     self.materiales = m
165
166   def proceso(self):
167     # Cabezara de la apariencia
168     print "\tappearance Appearance {"
169     for m in self.materiales:     #Proceso los materiales
170       mat = Material(m)
171       mat.proceso()
```

```
172     # Cierre de la apariencia
173     print "\t} # Fin apariencia"
174
175   class Malla:
176    def __init__(self, m):
177     self.malla = m.getData()
178     self.nombre = m.name
179     if (self.malla.getMode() & NMesh.Modes.TWOSIDED) == 0:
180       self.doblecara = 1
181     else:
182       self.doblecara = 0
183     self.verts = self.malla.verts
184     self.faces = self.malla.faces
185     self.matriz = m.matrixWorld
186
187    def proceso(self):
188     # Inicio el proceso de la malla
189     print "\tgeometry IndexedFaceSet {"
190     # Una o dos caras
191     if self.doblecara == 1:
192       print "\t\tsolid FALSE # Dos caras"
193     else:
194       print "\t\tsolid TRUE # Una cara"
195     self.vertices()
196     self.caras()
197     # Finalizo el proceso de la malla
198     print "\t} # Fin de la geometría:", self.nombre
199
200    def vertices(self):
201     print "\t\tcoord DEF c_%s Coordinate {" % self.nombre
202     print "\t\t\tpoint ["
203     for v in self.verts:
204       vr = self.absoluto(v)
205       print "\t\t\t\t %f %f %f," % (vr[0], vr[1], vr[2])
206     print "\t\t\t] # Final de los vértices"
207     print "\t\t} # Fin coordenadas c_%s" % self.nombre
208
209    def absoluto(self, v):
210     vr = Mathutils.Vector([v[0], v[1], v[2]])
211     vr.resize4D()
212     return Mathutils.VecMultMat(vr, self.matriz)
213
214    def caras(self):
215     print "\t\tcoordIndex ["
216     for f in self.faces:
217       verCara = ""
218       for v in f.v:
219         verCara += "%d, " % v.index
220       print "\t\t\t%s-1," % verCara
221     print "\t\t] # Final de la definición de las caras"
222
223   class Objeto:
224    def __init__(self, obj):
225     self.objeto = obj
```

```
226       self.nombre = obj.name
227       self.malla = obj.getData()
228
229     def proceso(self):
230         print "# Esto es el objeto malla: %s" % self.nombre
231         # Comienza el proceso
232         print "DEF %s Shape {" % self.nombre
233         # Obtengo los materiales
234         mats = self.malla.materials
235         # Proceso la "apariencia"
236         ap = Apariencia(mats)
237         ap.proceso()
238         # Proceso la "malla"
239         malla = Malla(self.objeto)
240         malla.proceso()
241         # Cierro la forma
242         print "} # Shape %s" % self.nombre
243
244   class Luz:
245     def __init__(self, l):
246       self.luz = l
247
248     def proceso(self):
249       print "# Aquí iría la definición de un punto de luz -"
250       print "# Dejamos al lector el ejercicio de definir los
                   distintos tipos de luces."
251       print "# Recordarle, únicamente, la conveniencia de
                   desactivar la luz 'subjetiva'"
252       print "#   que coloca VRML por defecto."
253       print "# ------------------------------------"
254
255   class Camara:
256     def __init__(self, cam):
257       self.camara = cam
258       self.nombre = cam.name
259       self.pos = cam.getLocation()
260       self.orien = cam.getEuler()
261
262     def proceso(self):
263       print 'DEF %s Viewpoint {' % self.nombre
264       print '\tdescription "%s"' % self.nombre
265       # Hago trampas con el ángulo de rotación del vector de
            orientación y escablezco un
266       # valor para evitar complicar más el script
267       print '\torientation', self.orien[0], self.orien[1],
                  self.orien[2], '1.5'
268       print '\tposition %f %f %f' % (self.pos[0], self.pos[1],
                  self.pos[2])
269       print '} # Fin de la definición de', self.nombre
270
271   class ExportadorEscena:
272     def __init__(self, nomFichero):
273       self.fichero = nomFichero
274
275
```

```
276   def proceso(self):
277     global losObjetos   #Objetos cargados en la lista global
278     # escribo la cabecera del fichero
279     print "#VRML V2.0 utf8\n"
280     print "#%s" % self.fichero # El nombre del fichero
281     for obj in losObjetos:      # Proceso la lista de objetos
282       tipo = obj.getType()      # Obtener el tipo de objeto
283       if tipo == 'Mesh':        # y tratarlo
284         o = Objeto(obj)
285         o.proceso()
286       elif tipo == 'Camera':
287         c = Camara(obj)
288         c.proceso()
289       elif tipo == 'Lamp':
290         luz = Luz(obj)
291         luz.proceso()
292
293   def fichero(nomFichero):
294     if nomFichero.find('.wrl', -4) < 0:
295       nomFichero += '.wrl'
296     # Abro el fichero y lo conecto con std
297     f = open(nomFichero, 'w')
298     std = sys.stdout
299     sys.stdout = f
300     # Proceso los datos
301     sce = ExportadorEscena(nomFichero)
302     sce.proceso()
303     # Devuelvo el control de stdout y cierro el fichero
304     sys.stdout = std
305     f.close()
306
307   #-----------------------------------------------------
308   # MAIN
309   #-----------------------------------------------------
310   try:
311     ARG = __script__['arg'] # Capturo argumento del submenú
312   except:
313     print "Versión antigua"
314     ARG = 'selec'
315
316   if Blender.Get('version') < 230:
317     print "Aviso: ¡Versión incorrecta de blender!"
318     print " No está utilizando Blender 2.30 o superior, "
319     print " descargue una versión más actualizada... "
320   else:
321     if ARG == 'todos':
322       escena = Scene.getCurrent()
323       losObjetos = escena.getChildren()
324     else:
325       losObjetos = Blender.Object.GetSelected()
326
327   Window.FileSelector(fichero, "Exportar")
```

Extendiendo Blender: Programando Plugins de Texturas

Guillermo Vayá



Si el modelado es la parte que nos permite comprender la forma del objeto, las texturas son todo aquello que "da color" a los modelos, tanto en el sentido figurado como en el estricto. Son imágenes aplicadas a las caras de los modelos, las cuales definen o modifican las propiedades del material.

Introducción a las Texturas y a las Texturas Procedurales

Las texturas pueden tener entre una y tres dimensiones, según queramos que afecten de forma lineal, superficial o a todo un volumen. Sin embargo, lo más habitual suelen ser las texturas bidimensionales, ya que con éstas se puede modificar todo un modelo aplicándol a las diversas caras.

Los parámetros que puede modificar son muy diversos. Para mucha gente, las texturas únicamente asignan el color a las caras, pero esto no es así, sino que un buen texturizado puede hacer brillar lo que antes no era más que un mal modelo. Parámetros como la reflexión o la transparencia *alpha* son ejemplos de algunas características del material que una textura puede modificar para una correcta visualización del objeto. Mediante las texturas podemos simular elementos de modelado que no existen ("bumping map", también llamado canal nor) o incluso generarlos ("displacement map").

Para poder conseguir nuestra biblioteca de texturas, hay que empezar distinguiéndolas en dos tipos:

- *Procedurales*. Son aquellas generadas por un algoritmo y que se implementan gracias a la programación. Son calculadas en tiempo de renderizado.

- *No procedurales*. Son dibujos, fotografías y, en definitiva, cualquier imagen estática. No dependen de ningún parámetro y, por lo tanto, siempre quedan de la misma forma.

Aunque aparentemente las no procedurales den la impresión de ser más fáciles de conseguir (mucha gente piensa que tomando una fotografía o haciendo un dibujo rápido se pueden generar buenas texturas), lo cierto es que tanto uno como otro modelo son bastante trabajosos a la hora de crearlos.

Una textura procedural se consigue mediante la aplicación de modelos matemáticos, es muy complicado generar un modelo matemático bueno que solucione el problema. Pero una vez está desarrollado éste, el paso al algoritmo programado es bastante sencillo. En cambio, en las no procedurales pasa justo lo contrario: encontrar lo que necesitas es fácil, pero su conversión a una imagen de ordenador que no tenga parámetros de iluminación residuales es realmente complicado.

Se ha de destacar que el principal problema y virtud de las texturas procedurales es su propiedad de mantener patrones repetitivos. Al aplicar una procedural como mapa de color, podemos ver cómo los caracteres predominantes se repiten de forma obvia, lo cual suele dar un desagradable efecto de irrealidad. Además, los patrones repetitivos son una de las causas más comunes de aliasing al alejarnos de la textura (o lo que es lo mismo, al reducirla).

De igual manera es una característica indispensable, ya que de otra forma no podríamos predecir el resultado: imagina una textura que a cada render tomase una forma distinta; sería poco útil, ¿no? Incluso aquellas que nos parecen completamente aleatorias no lo son, y no sólo porque un ordenador no pueda generar un elemento aleatorio puro, sino porque ya hemos visto que no interesa.

Todo esto nos lleva a pensar que las texturas procedurales son el aliado perfecto para generar modificaciones en los materiales.

Consiguiendo bmake

Los desarrolladores de Blender hacen todo lo posible por hacer más fácil la colaboración con los proyectos de la Blender Foundation. Un claro ejemplo es la herramienta `bmake`. `bmake` es un shell script creado para facilitar la tarea a aquellos interesados en desarrollar un plugin. Compilará nuestro código y generará una librería de enlace dinámico (generalmente caracterizada por la extensión `.so` en linux). De ésta manera los plugins que ejecutemos correrán en Blender como si fuesen parte del programa

original. Es por esto que hay que tener cierto cuidado al ejecutar plugins que no sepamos que son estables, cualquier error en el plugin podría estropear la ejecución de Blender y causar un cierre prematuro e indeseado de éste.

Si bien hasta aquí es todo muy bonito, el futuro desarrollador de plugins puede encontrarse con un pequeño contratiempo. En los paquetes de ciertas distribuciones (como por ejemplo en Debian) la utilidad `bmake` y todo lo necesario para la compilación no está empaquetado. Ésto puede ser debido a que realmente debería pertenecer a un futuro `blender-dev`.

Para conseguir los ficheros necesarios bajamos una versión del cvs:

```
# cvs -d:pserver:anonymous@cvs.blender.org:/cvsroot/bf-blender
   co blender
```

y la compilamos

```
# scons release
```

La utilidad `bmake` estará en `(path cvs)/blender/release/plugins/bmake` y las librerías necesarias en `(path cvs)/blender/source/blender/blender-pluginapi`. Copiaremos `bmake` en el directorio de plugins y los *headers* los compiaremos bajo `plugins`, en el directorio `include`.

En dicho directorio también hay otros dos llamados `texture` y `secuencer`.

En `texture` será donde crearemos los ficheros `.c` con el código del plugin.

¡Ahora ya tenemos todo! Para compilar iremos a la carpeta `textures` y teclearemos:

```
# ./bmake plugin.c
```

Para probarlo en la carpeta `textures` debería haber dos ejemplos: `tiles.c` y `clouds2.c`

```
# ./bmake tiles.c
# ./bmake clouds2.c
```

Cualquiera de los anteriores debería compilar sin problemas.

Estructura de un plugin de texturas

En la documentación de Blender.org viene una estructura básica para comenzar a diseñar un plugin. Aprovecharemos este recurso para evitar empezar de cero.

```
# include "plugin.h
```

Esta directiva de preprocesador ha de estar incluida en todo plugin que queramos desarrollar. Si no lo hacemos así, habrá varias constantes y funciones que no estarán disponibles y por tanto no se podrá compilar correctamente.

```
char name[24] = "";           /* Texture name */
```

Aquí irá el nombre de nuestro plugin tal y como queramos que aparezca en la interfaz de Blender, como se puede apreciar, tiene un máximo de 23 caracteres.

```
#define NR_TYPES 3
char stnames[NR_TYPES][16] = {"Intens", "Color", "Bump"};
```

`NR_TYPES` da el número de subtipos que existirán en nuestro plugin. De esta manera se puede tener varias texturas procedurales de características parecidas bajo un único archivo de plugin, pudiendo cambiar de una a otra desde la selección que nos brinda la interfaz de Blender. Un ejemplo de esto es la textura de Clouds, podemos seleccionarlas en blanco y negro o en color. La siguiente línea indica los nombres que asignaremos a estos subtipos. No se debe superar los 16 caracteres para el nombre (no, ni siquiera modificando la declaración, si haces eso Blender puede dar resultados inesperados).

```
VarStruct varstr[] = {
   {NUMIFLO, "Const 1", 1.7, -1.0, 1.0, ""},
};
typedef struct Cast {
   float a;
} Cast;
```

Estas dos estructuras son las que nos darán la información que le pidamos al usuario. Hay que tener en cuenta que han de estar ordenadas las variables de igual forma que los botones, para que el valor de uno sea asignado a la otra.

`varstr` define los botones que se van a mostrar así como sus parámetros básicos. Se podrían dividir en dos grupos básicos:

1. *Numéricos*: Está dividida en 3 subtipos combinados con un `flo` (coma flotante) o con un `int` (para enteros) mediante un operador `or` ("|"):
   - `num`: valor numérico, se edita directamente.
   - `numsli`: barra deslizadora para elegir el valor.
   - `tog`: un interruptor, consta de dos posiciones.
2. *Caracteres*: `label`, sirven para entradas de tipo texto.

A continuación especificamos el nombre, que será lo que aparezca en el texto del botón. No debe superar los 15 caracteres.

Para los elementos de tipo numérico, debemos especificar el valor por defecto, mínimo y máximo, en este orden. En el caso de los `tog`, el valor mínimo es el de pulsado y el máximo el de libre.

Detrás viene el "tip" o la ayuda que aparecerá cuando el puntero esté unos momentos encima del botón. El máximo es de 80 caracteres y si no se quiere usar, debemos dejar un `string` nulo ("").

El `Cast` es la estructura con la cual se nos pasarán los valores tomados por la interfaz. Por cada elemento que hubiese en el `varstr`, aquí habrá otro, en el mismo orden y con el tipo apropiado. Generalmente se suele usar el mismo nombre que para el botón, aunque esto último no es necesario (es una práctica recomendable, ya que mejora la legibilidad de nuestros programas y acelera la depuración de los mismos).

```
float result[8];
```

Aquí irá lo que queramos devolver a Blender. No tiene porqué estar relleno entero, sino que dependerá del valor que demos a la sentencia `return` al final de la función `texDoit`.

- `result[0]`: intensidad. Éste es el único que hay que rellenar siempre.
- `result[1]`: valor de rojo.
- `result[2]`: valor de verde.
- `result[3]`: valor de azul.
- `result[4]`: valor de alfa
- `result[5]`: desplazamiento de la normal en el eje X.
- `result[6]`: desplazamiento de la normal en el eje Y.
- `result[7]`: desplazameinto de la normal en el eje Z.

```
float cfra;
```

Esta variable nos la establece blender con el frame actual.

```
int plugin_tex_doit(int, Cast*, float*, float*, float*);
```

Prototipo de la función principal para registrarla en la interfaz de blender. Más adelante hablaremos de sus parámetros.

```
/* Fixed Functions */
int plugin_tex_getversion(void) {
   return B_PLUGIN_VERSION;
}

void plugin_but_changed(int but) { }
```

Esta parte del código no hace falta tocarla, y no deberemos tocarla a mnos que sepamos lo que estamos haciendo. Aun así, deberemos añadirla por compatibilidad de código.

```
void plugin_init(void) { }
```

Esta función sólo la modificaremos si hace falta realizar algún tipo de cálculo o inicialización antes de comenzar con el procedimiento. Es importante tener en cuenta que si este plugin se usa en más de un sitio, la función `init` será llamada cada vez que se acceda a uno de estos elementos, por lo que no se debe inicializar elementos de forma global a menos que estemos muy seguros de que es eso lo que queremos.

```
void plugin_getinfo(PluginInfo *info) {
  info->name=name;
  info->stypes=NR_TYPES;
  info->nvars=sizeof(varstr)/sizeof(VarStruct);

  info->snames=stnames[0];
  info->result=result;
  info->varstr=varstr;
  info->init=plugin_init;
  info->tex_doit=(TexDoit) plugin_tex_doit;
  info->callback=plugin_but_changed;
}
```

Esta función es la que se encarga de registrar los nombres de los distintos elementos de comunicación entre Blender y el plugin. Si cambias algún nombre más arriba, entonces también deberás cambiarlo aquí. Aunque lo más recomendable sería no cambiar ninguno de los nombres clave para que la gente pueda entender mejor el código.

```
int plugin_tex_doit(int stype, Cast *cast, float *texvec,
                    float *dxt, float *dyt)
```

Cabecera de la función principal. Cuando Blender quiera ejecutar el plugin, ésta será la función a la que llame (después de haber llamado a la de init si es la primera vez que accede al plugin).

Parámetros:

- `stype`: indica el subtipo que seleccionó el usuario, en el caso de haber más de uno.
- `cast`: estructura con los datos de la selección de parámetros del usuario a través de la interfaz de botones que le dimos.
- `texvec`: array de tres elementos, indicando la posición x, y, z del `texel` (el punto de la textura).
- `dxt` y `dyt`: estos valores estarán a `null` a menos que se haya pedido antialiasing (`osa`). En caso de contener algún valor, indicarán el tamaño de la textura en espacio de pixel (la explicación de este concepto queda fuera de este manual, consulta algún libro de texturas procedurales para saber lo que es).

```
if (stype == 1) {
   return 1;
} if (stype == 2) {
   return 2;
}
   return 0;
```

De todo lo anterior sólo explicaremos el valor de `return`. Según devolvamos 0, 1, 2 ó 3, Blender accederá a distintos valores del array `result`.

- 0: únicamente mirará la intensidad.
- 1: mirará tanto la intensidad como los valores de color `rgba`.
- 2: usará los valores de intensidad y de mapa de desplazamiento.
- 3: accederá a todo el array.

Ejemplo: Smiley.c

Ahora comenzaremos a desarrollar un sencillo plugin. Digo sencillo ya que no tiene ni siquiera antialiasing, sólo con este tema ya podría constituir por sí solo un tema entero debido a la cantidad de posibilidades que tiene así como la elección de cuál es mejor o peor según qué casos.

El plugin generará un “smiley” ligeramente configurable, podremos cambiar el tamaño así como si está triste o alegre (menuda chorrada, ¿no?). Con este plugin quiero demostrar que mediante procedurales se pueden realizar múltiples texturas, incluso aquellas más inverosímiles, aunque a veces no sean las más útiles. Además nos servirán como ejemplo para introducirnos un poco más en el mundo de las texturas procedurales de Blender.

Empezamos poniendo las librerías en las cuales se va a apoyar nuestro plugin. Como untilizaremos algunas funciones matemáticas avanzadas (seno, coseno y potencia) deberemos incluir la librería `math`. Además añadiremos la definición de plugin para que use los parámetros, funciones y constantes necesarias para la `api` de plugin.

```
#include <math.h>
#include "plugin.h"
```

Ahora le daremos un nombre, ¿qué mejor que smiley si eso es lo que es? Es conveniente siempre dar nombres que aseguren al usuario la comprensión de lo que hace el plugin a la vez que son cortos.

```
char name[] == "smiley";
```

Dado que esto es un ejemplo sencillo y no queremos complicarnos la vida, usaremos un único subtipo de plugin. Aquí podríamos haber metido tantos como posibles caras queramos tener, pero lo dejaremos en sólo uno. Le asignaremos el nombre smiley ya que no v a haber más tipos.

```
#define NR_TYPES 1
char stnames [NR_TYPES][16] = {"smiley"};
```

Ahora definimos los parámetros que queremos que sean modificables por el usuario, así como su presentación en la interfaz de Blender. En nuestro caso sólo queremos que pueda jugar con dos de ellos: el radio del smiley y la boca. Usaremos un deslizador (“slider” en inglés) para variar esos tipos. Se me ocurre que también podríamos haber usado un interruptor (“toggle”) para la boca, pero con un deslizador podemos dar un mayor rango de valores.

Para el radio elegimos un valor de 0 a 1, con 0 no habrá cara, mientras que con 1 ésta ocupará todo. Para la boca pondremos valores más pequeños, ya que los usaremos a modo de tanto por ciento e incluímos valores negativos y positivos para que exista tanto una cara sonriente como una cara triste.

```
VarStruct varstr[] = {
  NUMSLI|FLO, "radio", 1.0, 0.0, 1.0, "radio de la cara",
  NUMSLI|FLI, "boca", 0.25, -0.25, 0.25,
              "altura del centro de la boca"
};
```

Ahora hay que decirle a Blender con qué variables se corresponderán en nuestro código. Aprovecho para recordar que deberemos pasárselas en el mismo orden que en la definición de la interfaz par que blender no se líe.

```
typedef struct Cast
{
    float rCara;
    float aboca;
} Cast;
```

Las dos variables siguientes las definimos para la compatibilidad con el sistema de plugins de Blender. En `result` es donde se almacenarán los datos a devolver. Nosotros sólo usaremos los de intensidad y color, pero aun así hay que usar un array de 8 elementos. `cfra` lo definimos pero no lo usaremos (es decir se quedará con valor 0).

```
float result[8];
float cfra;
```

La siguiente parte es la parte que no vamos a tocas para nada. Ya que no nos sirve de mucho.

```
int plugin_tex_doit (int, Cast*, float*, float*, float*);
int plugin_tex_getversion (void)
{
  return B_PLUGIN_VERSION;
}

void plugin_but_changed(int but)
{
}
```

Ahora toca rellenar la función de inicialización del plugin. Como nuestra cara no necesita tener ningún valor inicializado, dejaremos la función en blanco.

```
void plugin_init(void)
{
}
```

La siguiente función es bastante importante, ya que indica a Blender la manera de comunicarse. Si bien hemos usado los nombres que venían por defecto en la estructura, aquí es donde habría que decirle si hemos usado otros nombres en vez de esos. Personalmente recomiendo seguir usando estos nombres, ya que así se hace más legilble para futuros desarrolladores formando una especie de estándar consensuado.

```
void plugin_getinfo(PluginInfo *info)
{
  info->name=name;
  info->stypes= NR_TYPES;
  info->nvars= sizeof(varstr)/sizeof(VarStruct);
  info->snames= stnames[0];
  info->result= result;
  info->cfra= &cfra;
  info->varstr=varstr;
  info->init= plugin_init;
  info->tex_doit= (TexDoit) plugin_tex_doit;
  info->callback = plugin_but_changed;
}
```

El siguiente trozo de código es el corazón y cerebro de nuestro plugin. Es el que hará todos los cálculos y llamará a las funciones necesarias. He de comentar que tiene un grave problema de diseño, para aquellos que ya hayan desarrollado más cosas con anterioridad verán que este pseudo-main no es modular, sino que realiza él solito y por sí mismo todos los cálculos. Cuan hubiera estado mejor separar el código en trozos más pequeños, invitando de esta manera a futuros desarrolladores a hacer pequeñas modificaciones y hacerles la vida más cómoda posible. Pero como esto no es un plugin pensado para su uso he preferido omitir el usar algunas funciones típicas de los lenguajes de shading así como el dividir la estructura en trozos. Otra de las razones por las que recomendaría usar la estructura en trozos es para poder utilizar el "antialiasing" de forma cómoda. De todas formas iremos analizándola poco a poco par que quede todo claro.

Primero usaremos la definición anterior del "main" del plugin y definiremos las variables que vayamos a usar así como su inicialización en caso de ser necesaria.

```
int plugin_tex_doit(int stype, Cast *cast, float *texvec, float
*dxt, float *dyt)
  float cox, coy, cara, boca, rboca;
  cara=cast->rCara;
  boca=cast->aboca;
  result[0]=1.0;              //precolocamos la intensidad
  result[4]=1.0;              //evitamos transparencias
  cox=texvec[0];
  coy=texvec[1];
```

Usando la fórmula básica del círculo comprobamos si el texel a tratar está dentro o fuera.

```
if ((pow(cox,2) + pow(coy,2))<pow(cara,2)) /* dentro de la cara */
```

He decidido dividir la cara en cuatro cuadrantes, para ello primero lo divido en dos mitades

```
if (texvec[1]>=0.0)           /* mitad superior */
```

Movemos la altura del pixel a un supuesto centro a la altura de los ojos.

```
coy=texvec[1]-(0.5*cara);
```

Ahora miramos en el lado al que está y modificamos la posición para hacerla relativa al centro del ojo en la coordenada X según corresponda.

```
if (texvec[0] <= 0.0)        /* cuarto superior izquierdo */
   cox = texvec[0] + (0.5*cara);      /* centro ojo izdo. */
else
   cox = texvec[0] - (0.5*cara);      /* centro ojo dcho. */
```

Con la misma fórmula del círculo, comprobamos si se está mirando dentro del ojo o fuera de él. Si estamos dentro del ojo, asignaremos un color negro y si estamos fuera, amarillo.

```
if((pow(cox,2) + pow(coy,2)) < pow((0.25*cara),2))  /*dentro*/
{
  result[1]=0.0; /*texel negro*/
  result[2]=0.0;
  result[3]=0.0;
}
else /*fuera del ojo*/
{
  result [1]=1.0; /*texel amarillo*/
  result [2]=1.0;
  result [3]=0.0;
}
```

Si no estábamos en la mitad superior, quiere decir que estamos en la inferior y por tanto sólo hemos de preocuparnos de la boca. Comprobamos que la posición pertenezca a la zona de la boca, ya que si no el color a tomar es bastante claro.

```
else /*mitad inferior*/
{
if ((texvec[0] >= -0.5*cara) && (texvec[0] <= 0.5*cara) &&
   (((texvec[1] >= -0.5*cara) && (boca >=0)) ||
   ((texvec[1]<= -0.5*cara)&&(boca<=0))))
...
}
```

Igual que antes modificamos las coordenadas para ponerlas en base al centro de la circunferencia que será la boca.

```
coy = texvec[1] + 0.5*cara + 0.25*boca;
cox = texvec[0];
```

Ahora ya solo queda por saber si estaremos dentro o fuera de ella. Como antes, si estamos dentro asignaremos un color negro y si estamos fuera, será amarillo.

```
if ((pow(cox,2) + pow(coy,2)) <= pow(boca*cara,2))
{
  result[1]=0.0;
  result[2]=0.0;
  result[3]=0.0;
}
else
{
  result[1]=1.0;
  result[2]=1.0;
  result[3]=0.0;
}
```

Si no estaba dentro del círculo de la cara, entonces pondremos el color como transparente, para que se vea el color original del material.

```
else
{
  result[4]=0.0; /* alfa al mínimo */
  result[1]=0.0;
  result[2]=0.5;
  result[3]=0.0;
}
```

Terminamos y salimos devolviendo el valor que indica que le pasaremos los valores de intensidad y color.

```
return 1;
```

Como se puede apreciar, este plugin es únicamente para un primer contacto con la interfaz de plugins, podría ser mejorable de muy diversas formas:

- Permitir la personalización de los colores o incluso admitir mezclas.
- Utilizar splines para la definición de la boca permitiendo así una mayor variedad de formas de la boca.
- Antialiasing.
- Soporte de canales *nor* o *displacement maps*.
- Utilización de otras primitivas para la definición de los elementos.
- Otros...

Notas y más información

Cosas a tener en cuenta:

1. Las coordenadas UVW usan un rango de coordenadas "[-1,1]".
2. Es preferible estar abriendo y cerrando Blender, ya que si no, puede no cargar bien.
3. Conviene tener un .blend con el plugin cargado. De esta forma se agiliza la visualización de los resultados si se está abriendo y cerrarndo Blender.
4. Al asignar un color no debemos olvidarnos del canal alfa, ya que en C las variables se inicializan a 0 y por tanto no veríamos más que el color base del material del objeto.

La compilación de plugins en Windows, se hace de manera distinta, ya que la herramienta `bmake` se basa en el `gcc` de linux (el cual no está disponible en windows). En cambio los comandos a ejecutar serán, aun cuando no es seguro que funcionen al "100%".

```
cd c:\blender\plugins\texture\sinus
lcc -lc:\blender\plugins\include sinus.c
lcclnk -DLL sinus.obj c:\blender\plugins\include\tex.def
implib sinus.dll
```

Algunos de los sitios de donde he sacado información y ejemplos de plugins:

- `http://download.blender.org/documentation/htmlI/c11605.html` pagina de la documentación oficial de Blender.
- `http://www.cs.umn.edu/~mein/blender/plugins` pagina con ejemplos y documentación de plugins.

El código del plugin completo

```
1   #include <math.h>
2   #include "plugin.h"
3
4   char name [] = "smiley";
5
6   #define NR_TYPES 1
7
8   char stnames [NR_TYPES][16] = {"smiley"};
9
10   VarStruct varstr[]={
11    NUMSLI|FLO, "radio", 1.0, 0.0, 1.0, "radio de la cara",
12    NUMSLI|FLO, "boca", 0.25, -0.25, 0.25, "altura del centro
                                        de la boca"};
13
14   typedef struct Cast
15   {
16     float rCara;
17     float aboca;
18   } Cast;
19
20   float result[8];
21   float cfra;
22
23   /***********Parte fija *********/
24   int plugin_tex_doit (int, Cast*, float*, float*, float*);
25   int plugin_tex_getversion (void)
26   {
27      return B_PLUGIN_VERSION;
28   }
29
30   void plugin_but_changed(int but)
31   {
32   }
33
34   void plugin_init(void)
35   {
36   }
37
38   void plugin_getinfo(PluginInfo *info)
39   {
40     info->name=name;
41     info->stypes= NR_TYPES;
42     info->nvars= sizeof(varstr)/sizeof(VarStruct);
43     info->snames= stnames[0];
44     info->result= result;
45     info->cfra= &cfra;
46     info->varstr=varstr;
47     info->init= plugin_init;
48     info->tex_doit= (TexDoit) plugin_tex_doit;
49     info->callback = plugin_but_changed;
50   }
```

```
51   /***********Fin parte fija *********/
52
53   int plugin_tex_doit(int stype, Cast *cast, float *texvec,
                         float *dxt, float *dyt)
54   {
55     float cox, coy, cara, boca, rboca;
56
57     cara=cast->rCara;
58     boca=cast->aboca;
59     result[0]=1.0; //precolocamos la intensidad
60     result[4]=1.0; //evitamos transparencias
61     cox=texvec[0];
62     coy=texvec[1];
63
64      if ((pow(cox,2) + pow(coy,2))<pow(cara,2))
         /*dentro de la cara*/
65       {
66         if(texvec[1]>=0.0) /*mitad superior*/ {
67           coy=texvec[1] - (0.5*cara);
68           if (texvec[0] <= 0.0) /*cuarto superior izquierdo*/
69             cox=texvec[0] + (0.5*cara); /*centro ojo izdo.*/
70           else
71             cox=texvec[0] - (0.5*cara); /*centro ojo dcho.*/
72           if((pow(cox,2) + pow(coy,2)) < pow((0.25*cara),2))
               /*dentro del ojo*/ {
73              result[1]=0.0; /*texel negro*/
74              result[2]=0.0;
75              result[3]=0.0;
76           }
77           else /*fuera del ojo*/ {
78              result [1]=1.0; /*texel amarillo*/
79              result [2]=1.0;
80              result [3]=0.0;
81           }
82         }
83         else /*mitad inferior*/ {
84           if ((texvec[0] >= -0.5*cara) && (texvec[0]
                 <= 0.5*cara) &&
85              (((texvec[1] >= -0.5*cara) && (boca >=0)) ||
86              ((texvec[1]<= -0.5*cara)&&(boca<=0)))) {
87                 coy = texvec[1] + 0.5*cara + 0.25*boca;
88                 cox = texvec[0];
89                 if ((pow(cox,2) + pow(coy,2)) <=
                       pow(boca*cara,2)) {
90                    result[1]=0.0;
91                    result[2]=0.0;
92                    result[3]=0.0;
93                 }
94                 else {
95                    result[1]=1.0;
96                    result[2]=1.0;
97                    result[3]=0.0;
98                 }
99           }
100
```

```
101             else {
102               result[1]=1.0;
103               result[2]=1.0;
104               result[3]=0.0;
105             }
106           }
107         }
108         else {
109           result[4]=0.0;/*alfa al minimo*/
110           result[1]=0.0;
111           result[2]=0.5;
112           result[3]=0.0;
113       }
114       return 1;
115   }
```

Rediseño de Yafray: Parte 1. La luz

Alejandro Conty Estévez
aconty@gmail.com ::

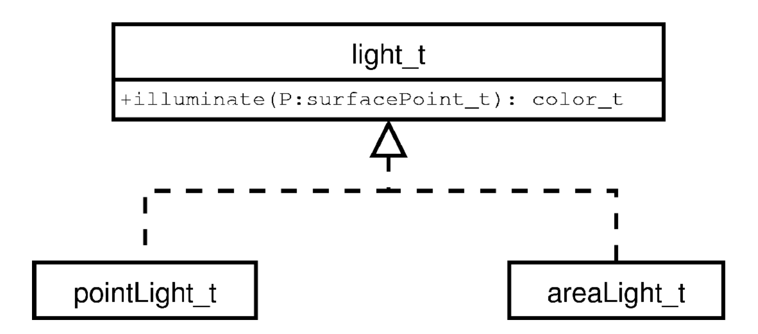


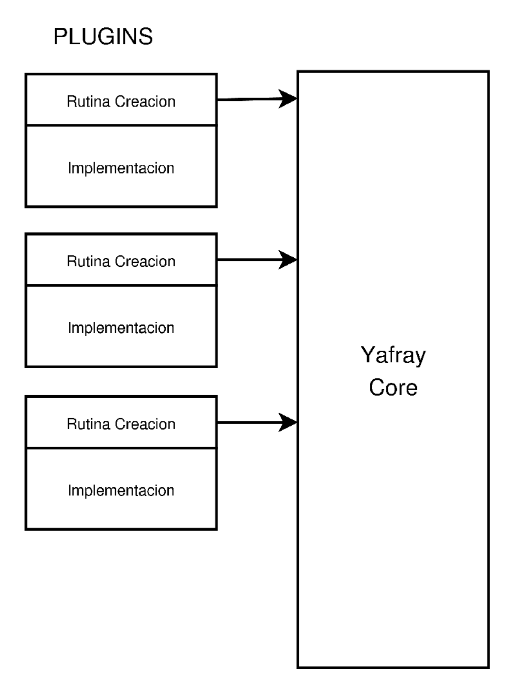
La luz en yafray se descompone en una serie de elementos en la escena llamados luces. La afirmación parece trivial, pero la singularidad reside en que no todos estos objetos se corresponden con lo que intuitivamente llamaríamos luz.

Estado actual

Antes de entrar en detalles, veamos el diseño de estos elementos. Cada luz en yafray consiste en un objeto abstracto capaz de iluminar un punto de la escena. Por ello, definimos un método virtual llamado *illuminate* que toma como argumento un punto de una superficie y devuelve el color reflejado en una dirección concreta.

Para devolver el color reflejado, es necesario que la luz llame al *shader* (material) asignado a esa superficie. Éste es uno de los puntos confusos del diseño sobre el que la gente a menudo pregunta. Parece poco intuitivo que la llamada a los shaders se delegue en las luces, por lo que volveremos sobre este punto más adelante. Por lo demás, con este interfaz, la lógica del cómputo de iluminación queda oculta tras esos objetos.





De esta forma podíamos separar el núcleo del render del cálculo de la iluminación. Definíamos la implementación de los distintos tipos de luces en ficheros separados sin que hubiera más dependencia que el citado interfaz. En una fase posterior del desarrollo, estos tipos de luces fueron separados aún más lejos con un sistema de plugins. Al ser tipos abstractos, sólo había que preocuparse de la creación de las instancias. A partir de ese punto, el objeto se manejaba de forma anónima y transparente.

Para la gestión de la creación de objetos se usaron factorías abstractas (ver patrón factory). El entorno de la escena disponía de una tabla de funciones de construcción indexadas por el tipo del objeto. No sólo para las luces, también para los shaders y fondos de escena. Éstas funciones factoría se diseñaron con un prototipo como el siguiente:

```
1   light_t * light_factory_t(paramMap_t &params,
    renderEnvironment_t &env);
```

Donde el parámetro “params” guardaba toda la configuración del objeto a crear. Todas estas factorías se indexaron en una tabla por nombre definida de la siguiente manera:

```
2   std::map<std::string,light_factory_t *> light_factories;
```

De la misma forma se hizo para las texturas, shaders, etc ... Cuando un plugin se carga, se llama a una función predefinida que éste ha de contener y que se encarga de registrar las factorías necesarias para todo lo que éste contiene. Veamos un ejemplo de código del plugin con los shaders básicos:

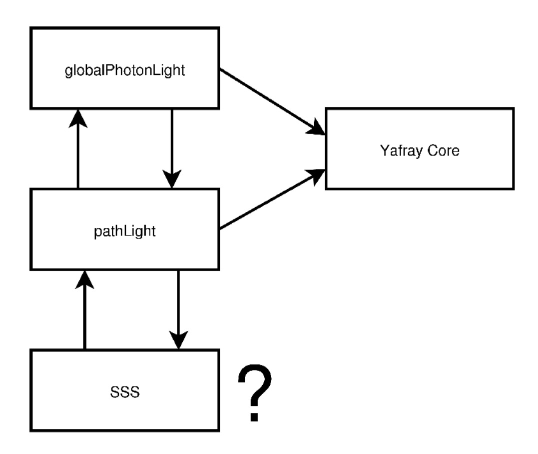
```
3  void registerPlugin(renderEnvironment_t &render)
4  {
5     render.registerFactory("generic",
                             genericShader_t::factory);
6     render.registerFactory("constant",
                             constantShader_t::factory);
7     std::cout<<"Registered basicshaders\n";
8  }
```

Aquí vemos como se registran las factorías para los tipos de shader "generic" y "constant". A partir de este momento, cuando el cargador encuentra una sentencia xml que ordena la creación un shader del tipo "generic" busca la factoría en la tabla y la invoca dándole todos los parámetros de la sentencia xml. Como resultado devuelve un puntero a un shader que se almacena en la escena para posterior uso.

Volviéndonos a centrar en las luces, todo este sistema permitió crear de manera independiente los tipos básicos de fuentes de luz sin tocar el núcleo. En general es un sistema bueno que sigue funcionando bien para los shaders. En las luces en cambio, no todo resultó tan bien. El problema fue que la definición de luz era demasiado amplia. Se entendía como luz cualquier cosa capaz de calcular la energía reflejada en una dirección sobre un punto dado. Debido a esto, caímos en el error de implementar métodos de iluminación global como luces en la escena. Éste es un caso claro de exceso de modularización, y es de estudio recomendado para todo el que se proponga diseñar un programa de envergadura.

Los dos primeros casos fueron la *hemilight* y la *photonlight*. La primera calculaba la luz proveniente del fondo de la escena y la segunda las cáusticas proyectadas desde un punto cualquiera de luz.

En ese momento, el concepto de luz se disolvió. Las luces pasaron a ser métodos de iluminación. Esto no hubiera sido mayor problema de no ser por la complejidad que alcanzaron ciertos métodos de iluminación. En concreto la "luz" *pathLight* usa complejos cálculos de iluminación indirecta, sistema de caché e incluso cáusticas. Para ello llega a usar incluso mapas de fotones que son creados por otra "luz" independiente. Esto crea una serie de dependencias molestas entre varias luces y el núcleo del render.



Como se indica en el gráfico, un supuesto módulo de subsurface scattering introduciría aún más dependencias. La única forma de evitarlo sería meterlo directamente dentro de la iluminación global, haciendo crecer la pathLight aún más. Ninguna de las dos opciones es muy recomendable.

Valoración del estado actual

Ante esta situación se pueden hacer las siguientes observaciones:

- Los plugins pierden sentido en el momento que aparecen dependencias camufladas y colisiones.
- El interfaz limitado con los plugins nos para de mejorar ciertos algoritmos y de añadir nuevas características.

El primer error viene de mezclar los conceptos de “fuente de luz” y “método de iluminación”. Las fuentes de luz son posiciones, superficies o volúmenes que emiten energía. ¿Qué se espera de ellas? Teniendo en cuenta los algoritmos usados en yafray, lo que se necesita de una fuente de luz se puede resumir en:

- Muestrear la fuente desde un punto cualquiera.
- Muestrear la fuente en cualquier punto de ésta y en cualquier dirección.

La primera de las funciones está pensada para el raytracing desde el punto de vista hacia las fuentes de luz, también conocido como raytracing estándar. En este proceso, para calcular la energía que llega a un punto se usan técnicas de montecarlo en las que se muestrea la fuente de luz para cada punto visible desde la cámara. Las luces puntuales son un caso particular de superficie nula en las que sólo una muestra es necesaria. En general, asumimos que todas las fuentes de luz son muestreables.

El segundo requisito es para poder llevar a cabo el photon mapping. Cuando hacemos esto recorremos el camino inverso, desde las luces hacia la escena. Por eso necesitamos una forma de obtener direcciones de disparo de fotones.

Interacción luz-shader

Al principio del desarrollo de yafray se partía del principio de que una fuente no puntual se muestreaba en diferentes direcciones y se llamaba al shader por cada una de ellas. Esto se debe a que originalmente, el resultado del shader depende no sólo de la energía que llega sino también de la dirección.

Uno de los típicos efectos que se consiguen con este sistema son las "*specular highlights*". Un "*fake*" de una reflexión borrosa de una luz puntual. Sin embargo, actualmente en yafray este método no se usa siempre. Por ejemplo en la iluminación global, donde el entorno se muestrea en la semiesfera tangente a la superficie. En lugar de llamar al shader para cada muestra, se calcula la energía total por separado y luego se llama al shader. Esto evita hacer tantas llamadas, y el resultado es lo suficientemente bueno como para que nadie se haya quejado aún.

Lo interesante es que si seguimos este principio de calcular la energía por separado y luego llamar al shader sin dirección alguna, el diseño se simplifica. Quitamos libertad al shader para hacer ciertos efectos. Pero dado que las reflexiones se calculan en yafray por separado, el único efecto comúnmente usado que perdemos es el de los brillos especulares de luces puntuales. No hay manera de simular este efecto por ninguna otra vía, así que trataremos de poner algún añadido al diseño para conservarlas.

Propuesta de diseño para fuentes de luz

Siguiendo la directiva anteriormente descrita de calcular la energía por separado y asumir que el shader no tendrá la libertad de hacer ningún otro tipo de cálculo experimental, el interfaz de una fuente de luz se reduce a calcular la energía que llega a un determinado punto.

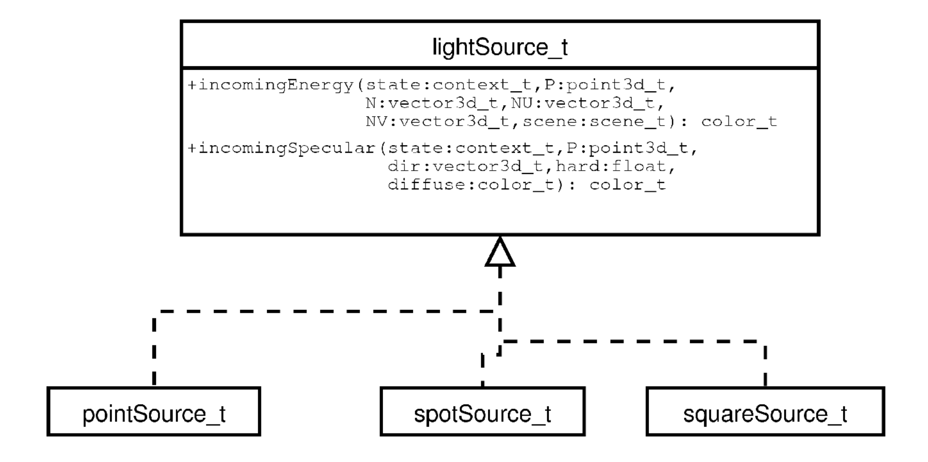
Con esto nos ahorramos el hacer un sistema de muestreo de fuentes de luz. Aún nos queda otro, el del mapeo de fotones, del que no podremos deshacernos. Pero ahora en vez de lanzar rayos en el núcleo en dirección a las fuentes, dejamos que sea la misma fuente la que haga el cálculo que tenga que hacer.

Para el caso de luces puntuales para las que pueda ser interesante un cálculo de luz especular, pondremos un método adicional que nos permita conservar dicha característica.

El primer método proporciona la manera estándar de calcular la cantidad de luz que llega a una superficie. Los parámetros son:

- P: El punto en cuestión.
- N: La normal al plano tangente al punto.
- NU y NV: Los dos vectores ortogonales que definen el plano tangente.
- scene: La escena.

El segundo de los métodos es un método Ad--hoc para las luces puntuales que puedan provocar reflejos especulares de ellas mismas. En el caso de luces no puntuales, la manera correcta de calcularlo es con una reflexión borrosa que se verá en una fase posterior de diseño.



Evidentemente, esta decisión de diseño que limita la libertad de un shader tiene un propósito. El método “incomingEnergy” encapsula toda la técnica montecarlo, y podemos aplicar ahí sin que afecte al resto una técnica de aceleración vía hardware gráfico. A continuación veremos cuales son las ideas detrás de este concepto.

Iluminación montecarlo vía GPU

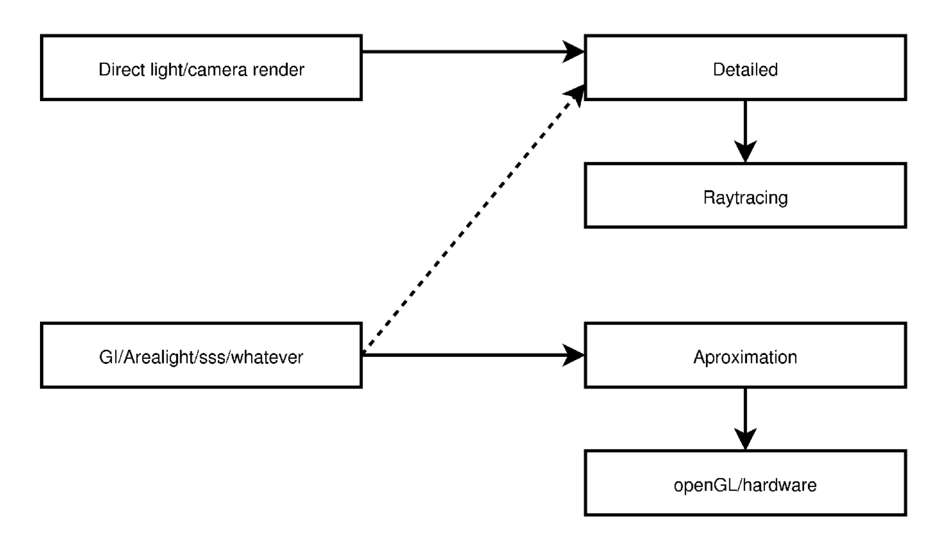
Existen varias formas en las que podemos aprovechar el hardware gráfico para acelerar el render. Recordemos que hay una fase de “*final gather*” en la que para cada punto de la escena visible disparamos rayos en la semiesfera tangente. Hacemos esto para simular una integral de toda la luz que llega a ese punto. Al fin y al cabo no deja de ser un render estocástico del entorno del punto que vamos a sombrear.

El propósito de este proceso es el de sumar toda la luz entrante. No es por lo tanto necesario que la calidad de ese render local sea muy alta. En realidad lo único importante es que el promedio de energía vista se aproxime a la real. Como es de esperar, para hacer un buen promedio es necesario lanzar bastantes rayos ralentizando el render.

Ahora supongamos que disponemos de una forma de renderizar una aproximación de la escena usando openGL por ejemplo. Se entiende que esta aproximación usa la iluminación del mapa de fotones mediante algún truco como texturas o colores en los vértices. Si disponemos de esta característica, en lugar de lanzar todos esos rayos en la semiesfera tangente, podemos simplemente hacer un render a baja resolución usando openGL.

La idea es renderizar con un ángulo muy abierto que aproxime la visibilidad de la semiesfera. Supongamos que hacemos un render de 100x100 pixels y luego hacemos la suma ponderada de esos pixels. Sería lo equivalente a tomar 10000 muestras por montecarlo estándar. Sólo que el hardware lo haría muchísimo más rápido. Teniendo en cuenta que el máximo razonable de muestras en yafray viene a ser unas 2000 ya que a partir de ahí empieza a ser excesivamente lento, el poder tomar 10000 muestras de golpe es sin duda un empujón bastante grande.

Además, no sólo este proceso puede beneficiarse del hardware, también el render de reflexiones borrosas, luces de área y cualquier otra cosa que pueda funcionar con una aproximación de la escena. En estos casos particulares lo que se hará es renderizar con un cono pequeño en lugar de usar uno de 180 grados.



Como se ve en el diagrama, siempre podemos tener ambas alternativas presentes. El render aproximado por hardware o el de calidad. Podemos elegir uno u otro a voluntad según los requisitos de calidad.

Tareas a completar

Para poder llegar a esto es necesario resolver los siguientes problemas:

- Recorrer el árbol BSP con un cono en lugar de un rayo.
- Poder renderizar en openGL con un cono de 180 grados sabiendo que ángulo corresponde a cada pixel del resultado.
- Poder renderizar triángulos con la información de color del mapa de fotones.

El último problema podemos aplazarlo ya que podemos avanzar dando pasos previos con sistemas de iluminación menos ambiciosos como hace la “hemiLight”.

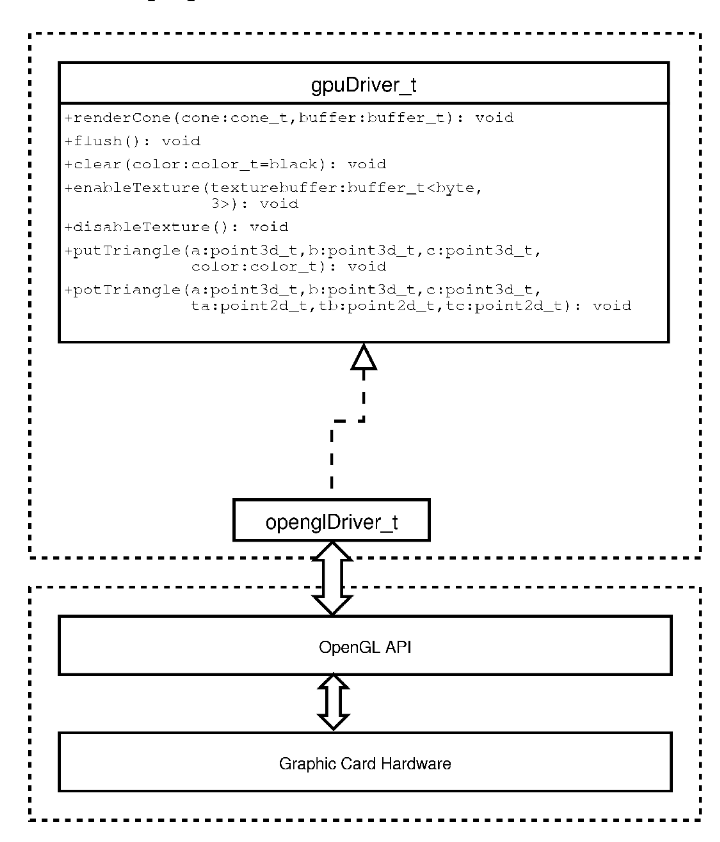
Para evitar volcar todos los triángulos de la escena en cada uno de esos renders locales, vendría bien poder escoger aquellos que caigan dentro del cono en el que vamos a renderizar. Hasta ahora cuando trazábamos un rayo, recorríamos los triángulos de la escena usando el árbol binario BSP. Eso evita tener que comprobarlos todos. Ahora queremos hacer lo mismo pero usando un cono en lugar de un rayo.

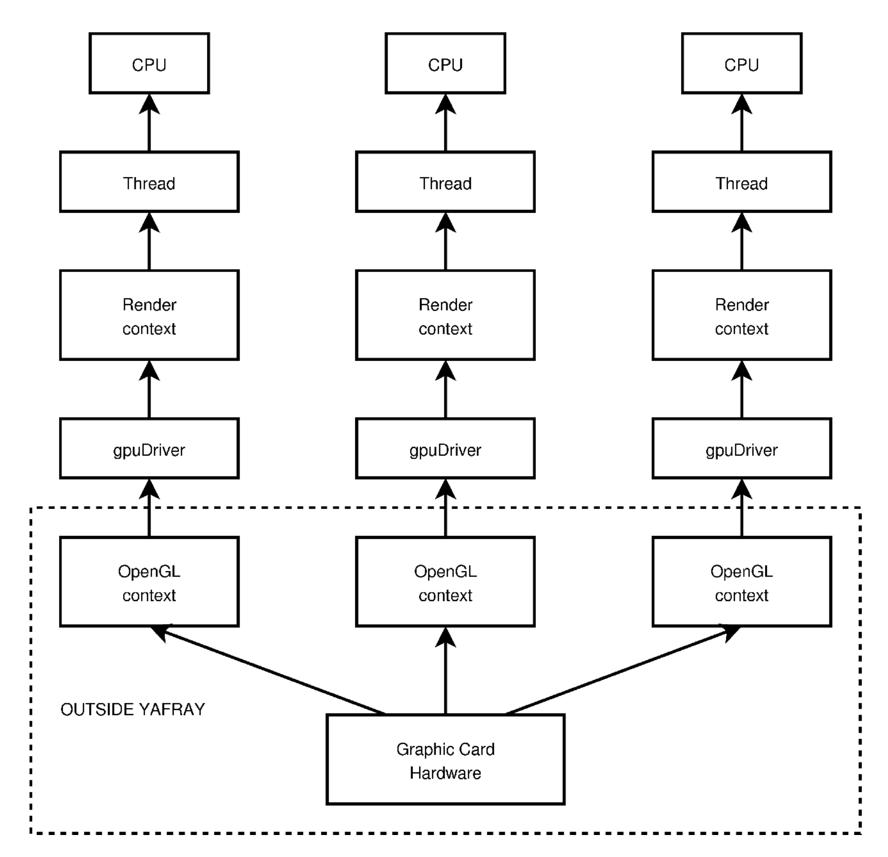
Afortunadamente este algoritmo es bastante sencillo y se puede alcanzar con algunas modificaciones sobre el algoritmo original para rayos. Añadiendo este método extra de recorrido al árbol ya tenemos una forma sencilla de esquivar los triángulos que no nos interesan.

En cuanto al segundo problema es sólo una cuestión de estudiar la proyección usada en openGL o el sistema que vayamos a usar. Para separar la interacción con el API de openGL, crearemos una clase de driver gráfico. En lugar de usar openGL directamente para el renderizado, lo haremos a través de este interfaz. En la siguiente sección veremos como podría ser.

Driver GPU

La funcionalidad que esperamos del hardware gráfico no es mucha. Simplemente querremos dibujar triángulos con un color o textura opcionalmente. El render se hará siempre sobre un buffer arbitrario. Veamos el interfaz propuesto:





La primera función, “renderCone” se llamará al principio del render local para elegir el cono de visión y el buffer donde se pondrá el resultado. Despúes se podrán usar el resto de funciones para pintar triángulos con un color plano o con una textura. Para las texturas, será responsabilidad del driver el mantener una tabla de correspondencias entre las texturas que el llamante está usando y las que vaya creando en el espacio del hardware gráfico.

Como vemos, se define un interfaz genérico de render simple del que luego derivamos a la implementación específica de openGL. Seguramente no usaremos otra cosa para renderizar vía hardware, pero de esta forma centralizamos el uso del API y descontaminamos el resto del código.

Existe un problema adicional en el uso del hardware. Yafray contempla la posibilidad de renderizar con hilos en varias CPU's. Hay que evitar que varios hilos accedan al hardware de manera concurrente. Afortunadamente openGL trabaja con contextos que se pueden intercambiar.

En yafray, cada hilo de renderizado posee un contexto que nada tiene que ver con el de openGL. En este contexto se guarda información de interés que es específica de cada hilo, como:

- Último objeto intersecado.
- Profundidad de render.
- Contribución actual al pixel que se está renderizando.
- etc ...

Lo mejor que podemos hacer es introducir un driver GPU en cada uno de estos contextos. De esta forma no se pisarán unos hilos a otros. Suponemos que el mismo driver se ocupa de crear su propio contexto openGL.

¿Por dónde empezar?

Antes de meternos directamente con la iluminación global total por hardware, conviene abordar problemas relajados relacionados con la misma. Si conseguimos hacer que funcione para los siguientes casos ya tenemos una gran parte del problema resuelta.

GPU hemilight

Veamos como podríamos calcular la iluminación que llega desde el fondo de la escena usando la GPU. El método de funcionamiento de la hemilight consiste en disparar rayos en una semiesfera tangente al punto de intersección. Si estos rayos golpean algún objeto se asume que son rayos de sombra y su contribución es nula. De otra forma alcanzan el fondo y los contamos como una contribución de luz positiva. Usamos el color del cielo como valor de contribución. Ya que estos son los primeros pasos, asumiremos que el color del cielo es constante. Más adelante se puede contemplar como hacer que podamos tener una textura de cielo o incluso un mapa HDR.

Para que una escena de cielo abierto tenga una acabado de calidad aceptable es necesario lanzar un número de muestras alrededor de 200. Si bajamos este número empieza a aparecer ruido en torno a las sombras. Evidentemente este método es costoso. Lo que se propone aquí es evitar ese proceso de lanzar 200 rayos. Para ello podemos intentar renderizar una aproximación del cielo visible desde dicho punto con la GPU. Dado que la contribución de los objetos a la luz es completamente nula, el proceso es sencillo. Veamos el siguiente pseudocódigo:

```
9   // P -> punto de interseccion
10  // N -> normal a dicho punto
11
12  gpudriver->renderCone(cone_t(P,N,180),buffer);
13  gpudriver->clear(background_color);
14
15  for each object in cone_t(P,N,180):
16  {
17    for each object face in cone_t(N,180):
18       gpudriver->putTriangle(face.a, face.b, face.c,
                                color_t(0));
19  }
20
21  gpudriver->flush();
22  color_t light=0;
23  for each pixel in buffer:
24  {
25    contri=cos(angle(pixel,N));
26    light+=contri*pixel;
27  }
28
29  return light/buffer.numPixels;
```

Como se ve, lo que hacemos es primer rellenar el buffer con el color del fondo. Acto seguido, lo que hacemos es pintar todos los triángulos de la escena de color negro. El resultado es un buffer en el que tenemos el cielo que puede verse desde ese punto. Si hacemos una media de todos los pixels, tenemos una aproximación de la luz que llega desde el cielo.

El problema de este método es que aproximadamente más de la mitad de las caras presentes en la escena tienen que volcarse para hacer el render. Cuando el número de caras en la escena es mucho mayor que el número de rayos necesarios para conseguir un resultado libre de ruido, es de esperar que la tarjeta gráfica lo haga más rápido que un método de raytracing. El problema está en lo que pasa cuando el número de caras es muy elevado. En ese caso habría que experimentar que es más rápido:

1. Lanzar 400 rayos contra 300000 caras.
2. Hacer un render de 20x20 de 300000 caras.

Es de esperar que 1 sea más rápido, sobre todo si dejamos crecer el número de triángulos de manera descontrolada. Pero habría que comprobar donde está el límite. También cabría estudiar si se podría trabajar con versiones decimadas de la escena. En resumen, el render de escenas con esta iluminación en la GPU es algo a experimentar.

GPU spherelight

Cuando la fuente de luz deja de ser algo tan grande como un cielo, las expectativas en cuando a usar la GPU mejoran. Si nuestra fuente de luz es una esfera de pequeño tamaño como podría ser una lámpara o una bombilla grande, el cono a renderizar disminuye.

Ya no es necesario volcar la mitad de los triángulos de la escena. Sólo aquellos que estén dentro del cono que une el punto a sombrear con la esfera que actúa de fuente de luz. En un punto alejado de ésta, el número de triángulos implicados es de esperar que se aproxime al número de muestras que queremos tomar o incluso que sea inferior.

Aquí es donde realmente nos podemos beneficiar del hardware gráfico. Un render de escasos polígonos sería mucho más rápido que lanzar 100 rayos contra la escena. En casos como estos podríamos permitirnos el lujo de tomar renders de 100x100, equivalente a 10000 muestras. Veamos un código de ejemplo:

```
30  // P -> punto de interseccion
31  // N -> normal a dicho punto
32  // C -> centro de la esfera
33  // R -> radio de la esfera
34
35  angulo=atan(R/distance(C,P));
36  direccion=C-P;
37  gpudriver->renderCone(cone_t(P,direccion,angulo),
                          buffer);
38  gpudriver->clear(color_t(0));
39  for each face in esfera:
40     gpudriver->putTriangle(face.a, face.b, face.c,
                              color_esfera);
41
42  for each object in cone_t(P,direccion,angulo):
43  {
44     for each object face in cone_t(P,direccion,angulo):
45         gpudriver->putTriangle(face.a, face.b, face.c,
                                  color_t(0));
46  }
47
48  gpudriver->flush();
49  color_t light=0;
50  for each pixel in buffer:
51  {
52    contri=cos(angle(pixel,N));
53    light+=contri*pixel;
54  }
55
56  return 4*light/(buffer.numPixels*M_PI);
```

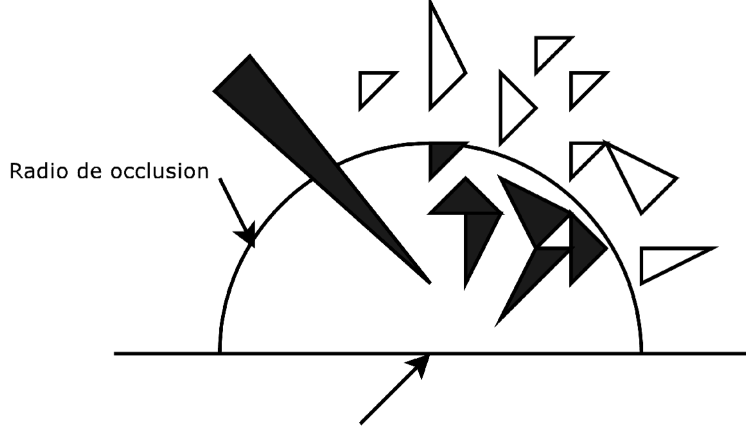
El render se hace siempre sobre buffers cuadrados, dado que el elemento emisor renderizado sobre él es un círculo, quedan puntos en negro que debemos ignorar. El factor 4/π del cómputo final se aplica para compensar esos puntos que en principio estamos contando como sombra cuando no lo son. Otra forma más eficiente sería esquivar de la suma esos puntos. Se ha puesto así por hacer el código más sencillo.

Este mismo método podría usarse para calcular reflexiones borrosas. Esto es comúnmente llamado “conetracing”, y sigue el mismo esquema el método de la esfera. La diferencia es que para tal cosa necesitaríamos una aproximación de la escena que incluya la iluminación, no servirían caras negras sin más.

GPU ambient occlusion

Éste es otro método que podría beneficiarse ampliamente del hardware gráfico. El ambient occlusion es un sistema muy parecido a la hemilight. La diferencia es que sólo se tienen en cuenta los triángulos que están a una distancia menor a un radio predefinido.

Es una forma de estimar la cantidad de luz ambiente que llega a un punto de la escena. Dado que ignora todo lo que esté más allá de cierto radio. Por este motivo es aún más falso de lo que produce la hemilight. No obstante, este método goza de gran popularidad entre los usuarios de Blender. Dado que el número de triángulos a volcar en el render está limitado por el radio predefinido, no sobrecargamos el hardware tanto como hacemos con la hemilight.



El código aproximado sería parecido al siguiente:

```
57  // P -> punto de interseccion
58  // N -> normal a dicho punto
59  // R -> radio de occlusion
60
61  gpudriver->renderCone(cone_t(P,N,180),buffer);
62
63  for each object in esfera(P,R):
64  {
65     for each object face in esfera(P,R):
66        gpudriver->putTriangle(face.a, face.b, face.c,
                                 color_t(0));
67  }
68
69  gpudriver->flush();
70  float occlusion=;
71  for each pixel in buffer:
72  {
73    occ=depth(pixel)/R;
74    if(occ>1) occlusion+=1; else occlusion+=occ;
75  }
76
77  return ambient_color*occlusion/buffer.numPixels
```

Se calcula la oclusión media en base a las distancias a los objetos que están en ese radio. El resultado se multiplica por el color de la luz ambiente y se devuelve. Nótese que asumimos que el driver gráfico nos puede proporcionar la profundidad de cada pixel renderizado.

Conclusión

Con este diseño del interfaz de las fuentes de la luz podemos implementar el cálculo de la energía en un punto de manera independiente. Esto es lo que permite plantearse el usar el hardware gráfico para estimar estas energías y ahorrar cómputo. Es posible que todo esto sea un esfuerzo en vano. Este documento no pretende ser más que un compendio de ideas a tener en cuenta para el futuro del desarrollo de yafray.

Modelado 3D basado en Bocetos

Inmaculada Gijón Cardós
Carlos González Morcillo
inmagijon@gmail.com · Carlos.Gonzalez@uclm.es ::



La fase de modelado de en proyectos 3D es dificil y tediosa en muchos casos, más aún cuando son necesarias superficies orgánicas. En la mayoría de las ocasiones, se parte de una primitiva y, aplicando transformaciones geométricas, se obtiene el objeto deseado.

Este proyecto presenta un prototipo de sistema que utiliza los trazos dibujados por el usuario para construir directamente una superficie tridimensional implícita. Nuestra aproximación difiere de las propuestas en trabajos previos (como Teddy de Takeo Igarashi) en que es un enfoque general y que implementa, con la misma aproximacion, la creacion de nuevos objetos y la operacion de extrusion (positiva y negativa).

Introducción

La forma general de construir objetos mediante modelado sólido es comenzar mediante una primitiva y, empleando operadores y transformaciones, construir un modelo modificado. En una primera clasificación, podemos hablar de modelado poligonal y modelado basado en superficies curvas.

Desde principios de los 90 se han desarrollado técnicas que permiten la construcción de modelos orgánicos de una forma rápida, como las superficies de subdivisión que comienzan con una *Malla de Control* que es subdividida recursivamente para obtener un resultado suave denominado *Superficie Límite*.

Otra aproximación es el uso de superficies implícitas (con trabajos previos en la construcción de una superficie que cubre un esqueleto). Nuestra aproximación se basa en la silueta dibujada por el usuario. En la figura 1 se muestra el interfaz de usuario del prototipo denominado MoSkIS (Modelling from Sketches Implicit Surfaces). En esta figura se calcula la espina dorsal del modelo que se emplea para situar las meta-elipses que formarán nuestro modelo.

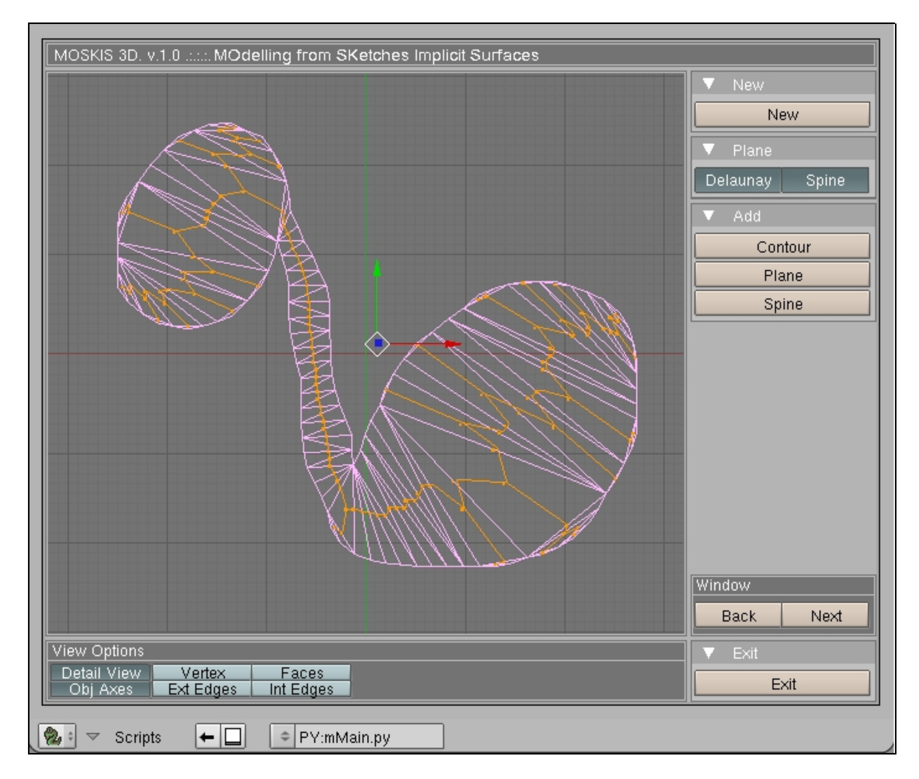


Figura 1. *Interfaz de usuario del prototipo MoSkIS 3D.*

Creación de un objeto

A continuación se hace una descripción del algoritmo que MoSkIS 3D desarrolla para llevar a cabo la generación de un objeto 3D a partir de un trazo libre realizado por el usuario.

Construcción de la silueta

El usuario dibuja un contorno mediante un trazo libre, el cual debe ser cerrado y no producir intersecciones consigo mismo, ya que en este caso el algoritmo fallaría.

A partir de este trazo se genera un polígono plano cerrado mediante la conexión del punto de comienzo y del punto final del trazo. Se identifican las aristas de este polígono mediante el ángulo mínimo que es requerido entre aristas consecutivas para crear una nueva arista.

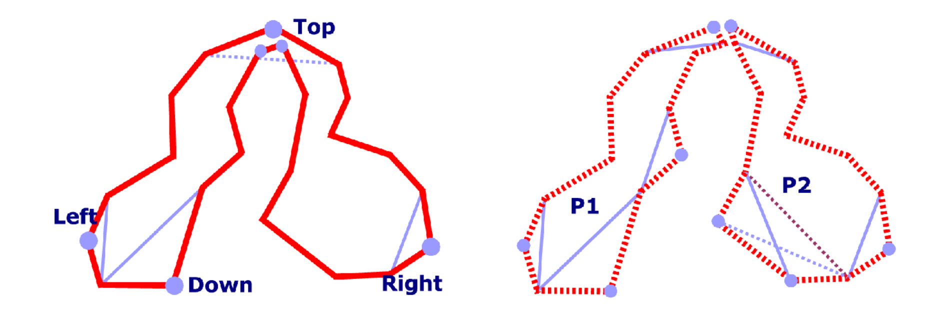


Figura 2. Izqda: *Vértices convexos a triangularizar.*
Drcha: *División del polígono en dos.*

Este parámetro es decidido por el usuario, al cual se le permite la opción de seleccionar el nivel de detalle con el que se construirá dicho polígono. A más detalle, más vértices y más coste computacional pero más exactitud en el objeto final obtenido.

El polígono que es generado a partir de este contorno dibujado, servirá como silueta del objeto 3D obtenido.

Obtención de la superficie

A la hora de obtener una superficie (compuesta de triángulos) a partir del polígono generado (ver figura 3 drcha) se realizan los siguientes pasos:

1. Triangularización básica: se realiza identificando sucesivamente los vértices convexos del polígono, concretamente cuatro, que son sobre los que se tiene la seguridad de ser convesos, ya que la figura puede ser tan complicada, que de lugar a errores fácilmente en esta identificación. Los vértices convexos de un polígono con toda seguridad, son los situados en los extremos, tanto del eje X, como del eje Y, es decir, el vértice superior, el vértice inferior, el vértice situado más a la derecha y el vértice situado más a la izquierda. Para cada uno de estos vértices se evalúa si es posible el trazado de la arista interna que va a formar un triángulo con las dos aristas que comparten el vértice detectado, y si es posible la construcción de dicho triángulo, ese vértice es eliminado del proceso de triangularización junto con las dos aristas que comparte, al estar ya triangularizada esa zona, con lo cual, el problema se va reduciendo.

Para que se lleve a cabo la construcción del triángulo, la arista interna a construir no puede cortar a ninguna otra arista del polígono (ver figura 2 Izqda.), es decir, no puede existir ningún vértice del polígono dentro del área del triángulo a construir ya que en ese caso la la triangularización sería errónea.

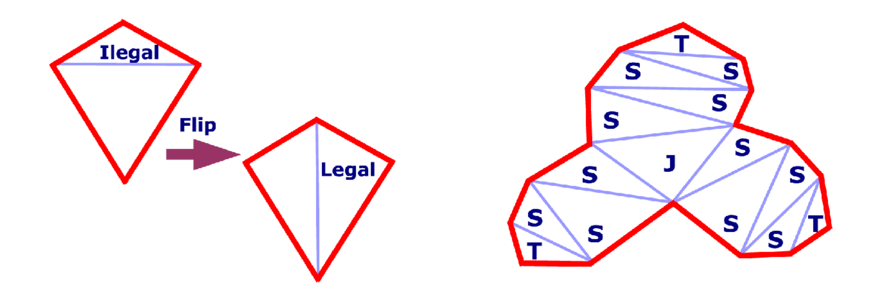


Figura 3. Izqda: *Flip de arista ilegal.*
Drcha: *Resultado de Delaunay con la catalogación de los tres tipos de triángulos.*

A la hora de construir un triángulo, se tiene que asegurar que la arista no corta a ninguna otra arista del polígono (ver figura 2 Izqda.), es decir, que no hay parte del polígono dentro de la superficie del triángulo a construir, ya que en ese caso, se alteraría la figura al no generar los triángulos en la superficie que encierra el polígono.

Como se puede ver en la figura 2 Izqda., no sería válida la arista interna que se ve punteada en el vértice superior. Para solucionar esta situación, se traza una arista interna que va desde este vértice hasta el más próximo a él de los que permanecen en interior del triángulo que se pretendía formar. Esta arista divide el polígono en dos (ver figura 2 Drcha.) y de esta forma se sigue aplicando una triangularización básica a cada uno de los polígonos (se divide el problema).

En la figura 2 Drcha. se observa también que el vértice inferior detectado en el polígono P2, aunque sea convexo y no contenga parte del polígono en el área que del triángulo a trazar, choca con otra arista interna de la triangularización, este caso nunca se da porque si ya se ha trazado la arista interna para formar un triángulo con las aristas que comparten el vértice situado más a la izquierda, este vértice y aristas se han eliminado del proceso de triangularización, con lo cual, la arista válida es la que vemos en color rojo.

De esta forma queda triangularizada toda la superficie limitada por el polígono cerrado generado.

Se denominan *aristas externas*, a las aristas que definen el polígono o silueta generada y *aristas internas*, a las aristas que han sido generadas mediante triangularización (ver figura 3 Drcha).

2. Delaunay: a la triangularización básica generada, se le aplica la optimización de Delaunay, consistente en la identificación y sustitución de aristas internas *ilegales* por aristas internas *legales*.

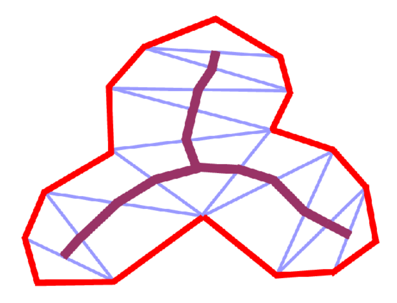


Figura 4. *Espina Dorsal.*

Una arista es *legal* cuando maximiza el ángulo mínimo del triángulo formado, y es *ilegal* en caso contrario. Si se identifica una arista *ilegal*, se realiza un giro (también denominado *flip*) de esa arista, por la opuesta en el cuadrilátero formado por los dos triángulos que comparten dicha arista.

De esta forma se asegura que la superficie del polígono quede repartida mediante triángulos de la forma más homogénea posible (ver figura 3 Izqda.).

Los triángulos generados a partir de la triangularización se pueden catalogar como (ver figura 3 Drcha.):

- **T-triangles**: compuestos de dos aristas *externas*.
- **S-triangles**: compuestos de una arista *externa*.
- **J-triangles**: compuestos de tres aristas *internas*.

Detección y elevación de la espina dorsal

Una vez generado el plano, se procede a detectar la *espina dorsal* (*spine*) de éste. Para ello se detectan los puntos principales del eje mediano de la forma inicial, a lo que llamaremos *puntos de conexión* de la espina dorsal. La unión meidante aristas de estos puntos de conexión me dan la espina dorsal del plano que representa la silueta del objeto.

Estos puntos de conexión se calculan recorriendo cada uno de los triángulos que forman el plano y analizando cada uno de ellos como se explica a continuación (ver figura 4):

- **T-triangle**: se detecta un punto de conexión en el centro de la arista interna del triángulo. Este punto también será detectado al evaluar el otro triángulo que comparte esta misma arista interna, y si es un S-triangle o un J-Triangle unirá este punto de conexión con el que corresponda según se explica a continuación.

- **S-triangles**: se detectan dos puntos de conexión, uno por cada una de las aristas internas del triángulo y serán situados en el centro de cada arista. Estos dos puntos de conexión serán unidos por una arista para formar la espina dorsal.

- **J-triangles**: se detectan cuatro puntos de conexión, uno por cada una de las aristas internas, situados cada uno en el punto medio de las correspondientes aristas, y un punto de conexión situado en el centro del triángulo. Cada punto de conexión pertenecientes a las aristas internas serán unidos mediante una arista de la espina dorsal al punto de conexión del centro del triángulo.

Una vez que todos los puntos de conexión han sido unidos mediante aristas formando la espina dorsal del plano, se procede a la elevación de la espina (ver figura 5).

El problema es computar la distancia desde la espina dorsal al límite del polígono. Lo ideal es construir secciones representativas exactas de la forma, en cada uno de los vértices de la espina dorsal, cuyo coste es bastante elevado. Por ello se calcula, para cada punto de conexión, dependiendo de si está asociado a una arista interna o está situado en el centro de un triángulo.

- Para el caso en el que los puntos de conexión sean punto medio de aristas internas, se calcula un promedio de las distancias entre los puntos de conexión y los vértices externos (cualquier vértice del plano) que están conectados directamente a ese vértice, y esa será la altura de ese punto de la espina.

- Para el caso en el que los puntos de conexión son el punto central de un J-triangle, se calcula como promedio de las distancias de elevación de los puntos de conexión directamente conectados a éste.

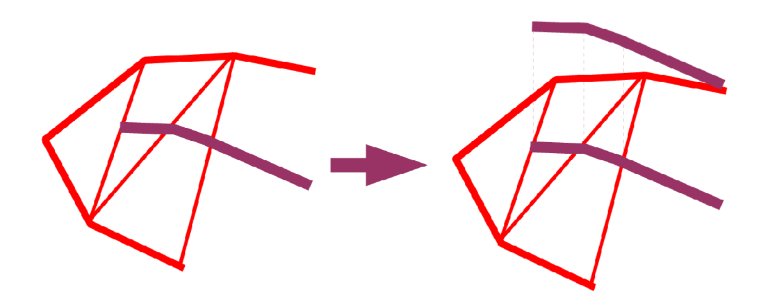


Figura 5. *Elevación de la espina dorsal.*

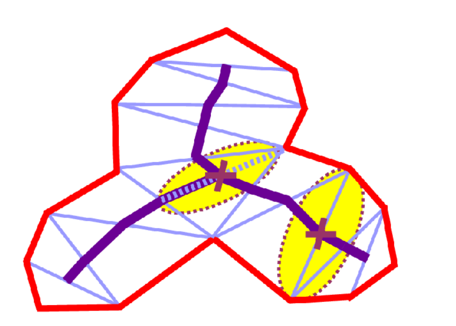
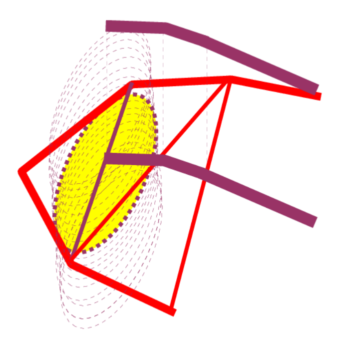



Figura 6. Izqda: *Dimensiones X e Y y orientación de cada metaelipse.*
Drcha: *Dimension Z de cada metaelipse.*

Obtención del volumen mediante metasuperficies

La obtención del volumen se realiza añadiendo para cada vértice o punto de conexión de la espina dorsal un metaelemento, concrétamente una metaelipse con una dimensión y orientación determinada (ver figura 6).

- La **dimensión X** de cada metaelipse se corresponde para el caso en el que su punto de conexión asociado es el punto medio de una arista interna, con la longitud de dicha arista interna; si por el contrario, el punto de conexión asociado se corresponde con el punto medio de un J-triangle, la dimensión X se corresponde con la longitud media de las bisectrices del J-triangle.

- La **dimensión Y** de cada metaelipse se corresponde con la distancia media desde ese punto de conexión al resto de puntos de conexión directamente conectados al mismo.

- La **dimensión Z** de cada metaelipse se corresponde con el doble de la altura de la espina en el punto de conexión asociado a la metaelipse, ya que la figura resultante va a ser simétrica, y tendrá una componente Z positiva de valor el de la altura de la espina y una componente Z negativa del mismo valor absoluto (ver figura 6 Drcha.).

Con este algoritmo se obtiene una malla 3D creada a partir de la unión de esas metaelipses añadidas, consiguiendo el objeto 3D con las dimensiones correspondientes a la forma inicial pintada por el usuario.

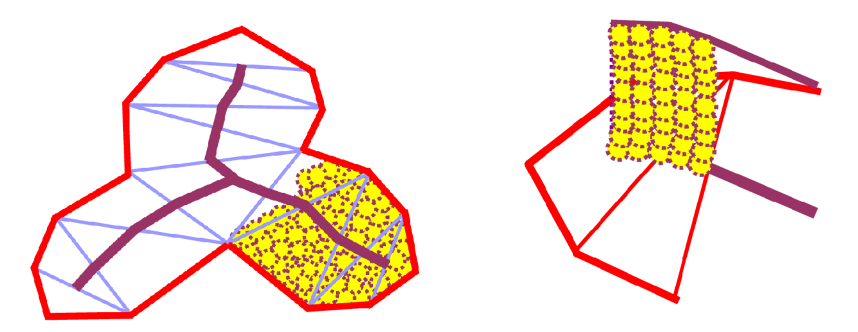


Figura 7. Izqda: *Metaelipses de dimensión fija y reducida.*
Drcha: *Metaelipses a lo largo del eje Z.*

Mejoras en la obtención del volumen

En la generación del objeto 3D según el algoritmo de generación de volumen especificado en el apartado anterior surgen dos problemas principales:

- Cuando en la generación de la espina dorsal, aparecen por ejemplo dos puntos de conexión muy distantes, pero que a sus otros lados respectivamente tienen puntos de conexión muy cercanos, como la dimensión Y que se establece para cada metabola es la media de las distancias, pueden aparecer huecos o cavidades en la malla 3D del objeto resultante debido a la falta de metaelementos en esa zona.

- En algunas figuras creadas por el usuario, se identifican saltos bruscos entre las metasuperficies, cuando en realidad, no se debía poder identificar visualmente cada metasuperficie, sino que se debería obtener una malla suavizada como resultado del trazo realizado originariamente por el usuario. Estos saltos son más destacados a cuánto menor es el nivel de detalle. A mayor nivel de detalle en la generación del polígono a partir del trazo del usuario, más vértices otenemos, por lo tanto más triángulos son generados en la triangularización y más puntos de conexión obtendremos en la construcción de la espina dorsal, con lo cual las dimensiones de los metaelementos cambian de uno a otro más progresivamente.

La solución a estos problemas pasa por la creación de metaelipses de dimensiones fijas reducidas, en vez de una por cada punto de conexión de la espina dorsal, crear las necesarias para cubrir todas las dimensiones (ver figura 7 Izqda.).

- Se sitúan metaelipses con dimensiones fijas y reducidas a lo largo de cada arista interna.
- Se sitúan metaelipses a lo largo de la mayor bisectriz de cada J-triangle.
- Se sitúan metaelipses a lo largo del eje de la espina dorsal, y cada cierta distancia, en ausencia de aristas internas, se crearán aristas internas imaginarias, con el mismo comportamiento que el citado anteriormente para la situación de metaelipses. Con esta decisión, se evita la generación de huecos indeseados en el objeto 3D generado.
- Se sitúan metaelipses cubriendo la distancia media desde el punto de conexión con los puntos conectados directamente.
- Se sitúan metaelipses a lo largo del eje Z cubriendo la altura de la espina dorsal, estableciendo una altura incremental desde la altura de un punto de conexión al siguiente. Así evitamos escalones bruscos en la malla 3D generada.

Como se puede ver en la figura 7 Drcha., (la figura ha sido simplificada viéndose únicamente metaelipses a lo largo de la espina dorsal para mayor claridad), se muestra hasta dónde llegarían las metaelipses a lo largo del eje Z, estableciendo como se ha dicho anteriormente un promedio entre la posición (en el eje Z) de las que están situadas en la posición de los puntos de conexión que las limitan.

Algoritmo de extrusión

A continuación se describe el algoritmo para la ejecución de la operación de extrusión sobre un objeto 3D previamente creado. Este algoritmo aún se encuentra en desarrollo en el sistema MOSKIS3D.

Trazo del usuario

En primer lugar, el usuario realiza un trazo seleccionando la superficie del objeto 3D a extruir, ese trazo, también llamado *trazo base*, debe formar una superficie cerrada. A continuación, realiza otro trazo, llamado *trazo de extrusión*, con la forma que desea conseguir con la operación (ver figura 8 Izqda.).

El algoritmo de extrusión, crea una nueva malla 3D basada en el *trazo base* creado por el usuario y en el *trazo de extrusión*.

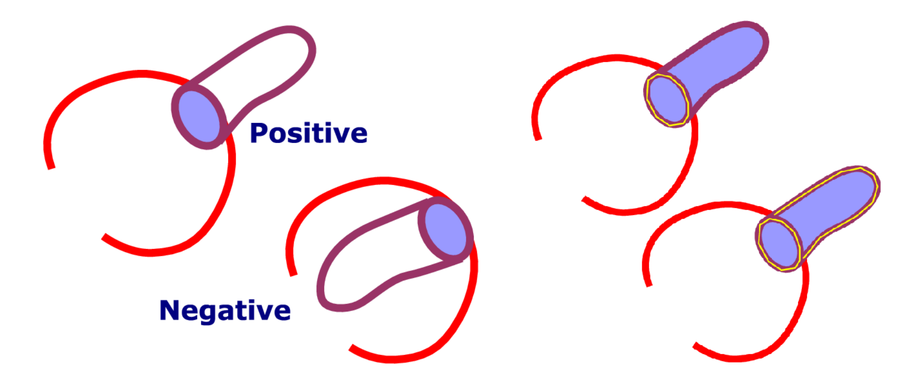


Figura 8. Izqda: *Trazos del usuario al realizar la operación de extrusión.* Drcha: *Polígono triangularizado.*

El usuario puede desear una *extrusión positiva*, es decir, una prolongación del objeto 3D hacia el exterior siguiendo la forma del trazo realizado; o una *extrusión negativa*, es decir, una concavidad en el interior del objeto 3D con la forma indicada (ver figura 8 Izqda.).

A partir del *trazo base*, se construye un polígono plano cerrado con el mismo nivel de detalle que el establecido para la generación del objeto 3D previo.

A partir del *trazo de extrusión*, también se genera un polígono plano, pero abierto, que tendrá el mismo nivel de detalle que en el caso anterior (ver figura 8 Drcha.).

Triangularización y generación de la espina

Una vez obtenido el polígono del trazo base, se procede a la triangularización de ese polígono cerrado, al igual que se hacía en el caso de creación de un objeto nuevo. Primero se triangulariza mediante un algoritmo de triangularización básica y a continuación se procede a una optimización por Delaunay (ver figura 3 Drcha).

Una vez triangularizado el trazo base, se tiene una superficie compuesta de los tres tipos de triángulos citados anteriormente T-triangles, J-triangles y S-triangles (ver figura 3 Drcha).

En este momento, se calcula la espina dorsal del plano generado a partir del trazo base, evaluando cada uno de los triángulos como se hacía en la detección de la espina para el caso de creación de un objeto nuevo (ver figura 4), obteniendo una serie de puntos conectados formando el eje central de la superficie seleccionada por el usuario, es decir, la superficie a extruir.

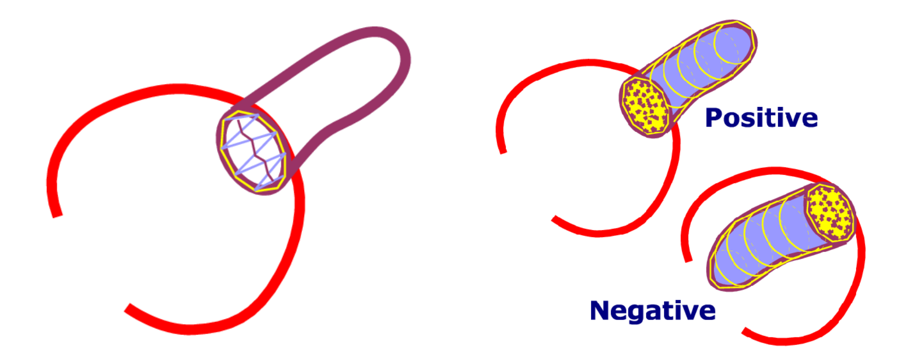


Figura 9. Izqda: *Trazo base triangularizado y espina dorsal detectada.*
Drcha: *Anillos de extrusión.*

En la creación de un objeto nuevo, el siguiente paso era la elevación de la espina dorsal mediante el cálculo de un promedio en cada uno de los puntos de conexión de la espina. En el algoritmo de extrusión, por el contrario, no se quiere crear un objeto 3D a partir de la forma creada por el usuario como trazo base, sino extruir esa forma a lo largo del trazo de extrusión, por lo que no tiene sentido el cálculo del objeto 3D asociado a esa forma, y por tanto, no tiene sentido el paso de elevación de la espina dorsal (ver figura 9 Izqda.).

Generación de la extrusión

Una vez obtenido el plano correspondiente al *trazo base,* y la espina dorsal de dicho plano, se procede a rellenar el plano con metaelipses.

Se sitúan metaelipses con dimensiones fijas y reducidas a lo largo de cada arista interna, para cada triángulo, y para el caso en el que se tenga un J-triangle se sitúan metaelipses a lo largo de la mayor bisectriz del triángulo.

Para evitar la generación de huecos indeseados, se sitúan metaelipses a lo largo del eje de la espina dorsal, y cada cierta distancia, si no existen aristas internas, se crearán imaginarias, con el mismo comportamiento. También se sitúan metaelipses cubriendo la distancia media entre el punto de conexión y los puntos conectados directamente a este.

Con este paso obtenemos un conjunto de metabolas con la forma de la superficie que delimitaba el trazo base generado originalmente por el usuario. Para generar la extrusión del *trazo base* a lo largo del *trazo de extrusión* se procede a la proyección del conjunto de metabolas, de forma que cada una de las proyecciones resultantes queden situadas perpendicularmente con respecto al trazo de extrusión y se redimensionen para ajustarse al tamaño del trazo.

El número de proyecciones en principio depende del nivel de detalle con el cual se haya generado el polígono abierto correspondiente al trazo de extrusión creado por el usuario. En los casos en los que la distancia entre los vértices del polígono (del trazo de proyección) sea muy grande, para evitar el problema de zonas huecas o poco pobladas del que hablábamos en el apartado de *Mejoras en la obtención del volumen*, delimitaremos la distancia máxima que debe haber para una nueva proyección, y en el caso en el que no se cumpla, se añadirá una nueva proyección (ver figura 9 Drcha).

Hay que diferenciar el caso en el que la extrusión es positiva o negativa, ya que en el primer caso las metaelipses generadas deben tener una atracción positiva, mientras que en el caso en el que la extrusión es negativa, se producirá una repulsión, para conseguir la concavidad o hueco que se desea (ver figura 8 Izqda).

Resultados

Empleando el método explicado en este artículo hemos obtenido el modelo de la figura 10. La silueta original fue convertida a 431 superficies implícitas (incluyendo las necesarias para realizar las extrusiones). El siguiente paso fue convertir la superficie a malla poligonal (formada por 12403 caras). Finalmente, el modelo fue suavizado y se aplicó un algoritmo de eliminación de aristas.

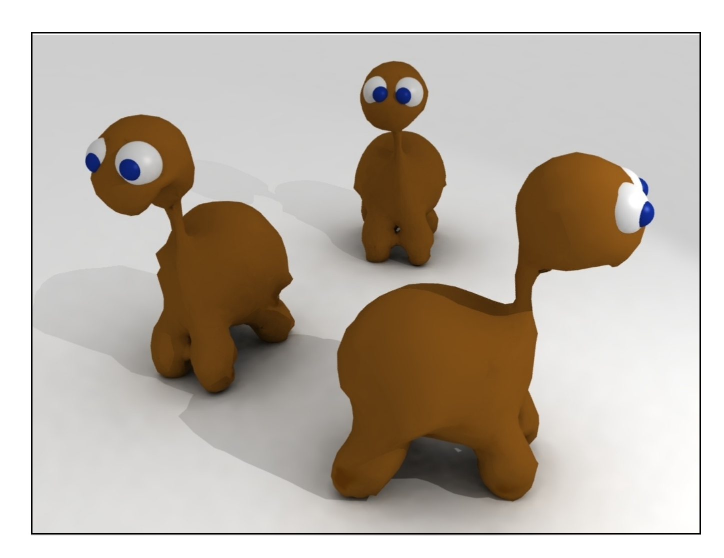


Figura 10. *Ejemplo de personaje realizado con MoSkIS 3D.*

Ciudad Real
Junio 2006